Edition 2022

Multiple authors

Frontend Unicorn

Learn code faster, be more efficient, become
a high-valued developer and earn more money

Table of Contents

INTRO

1. Who is this book for? 7

2. Frontend Unicorn 11

3. Stack decisions 29

4. Understanding design 46

5. You're the designer! 119

6. Nudge the designer 138

LEVEL 1

7. Semantics & Accessibility 155

8. Code = business card 173

9. Back-end basics 190

10. Code =/= Product 208

11. Developer in a startup 218

12. Master GIT 230

FIRST BOSS

13. Work/Life balance 245

14. Taking notes 255

15. Time management 267

LEVEL 2

16. Code review 279

17. How to learn effectively 289

18. Learn something rare 305

19. The mobile world 316

20. Impostor syndrome 327

21. Details that make a difference 338

FINAL BOSS

22. Debugging 359

23. Communication skills 373

24. Useful tools 386

25. How to get a raise 401

26. Thank you 418

Hello! Szymon, Albert, Albert & Michal here.

Welcome to "Frontend Unicorn"!

A couple of years ago we noticed, that as the front-end development industry becomes more and more popular, many people are repeating the same mistakes over and over. That led to junior developers competing in an unfair battle for a cool job, where hourly rates are the only advantage. Screw that! Unicorns exist and by learning the right approach, you can become one too. So just after writing our super-popular UI design book (www.designingui.com) we decided to tackle the front-end next. Creating popular educational materials and speaking at conferences all across Europe allowed us to distill the best of the best practices for you. **Let's go!**

In 2013 we co-founded HYPE4 - a design-driven software house. Since then, we've been working for both small startups and Fortune-500 companies from all over the world. We were lucky enough to be able to work on nearly every product category (from banking, finance, advertising, and media to pregnancy tracking).

We started as a small, design-only studio with three employees.
We did the designs, and then external software houses handled the development. We quickly noticed a huge problem with that approach. The execution quality was so low that we were rarely satisfied.
In some cases, the designs were butchered beyond recognition.
We decided to take action by starting the development branch of the company but strictly following our design-driven approach.

Our goal was always to work with true unicorn developers - people who contribute to the project a lot more than 80%-precise execution.

At HYPE4, we build web and mobile products of all sizes. After hours we also work on our internal startups and have a lot of fun doing it.
If you'd like to learn more, go to:

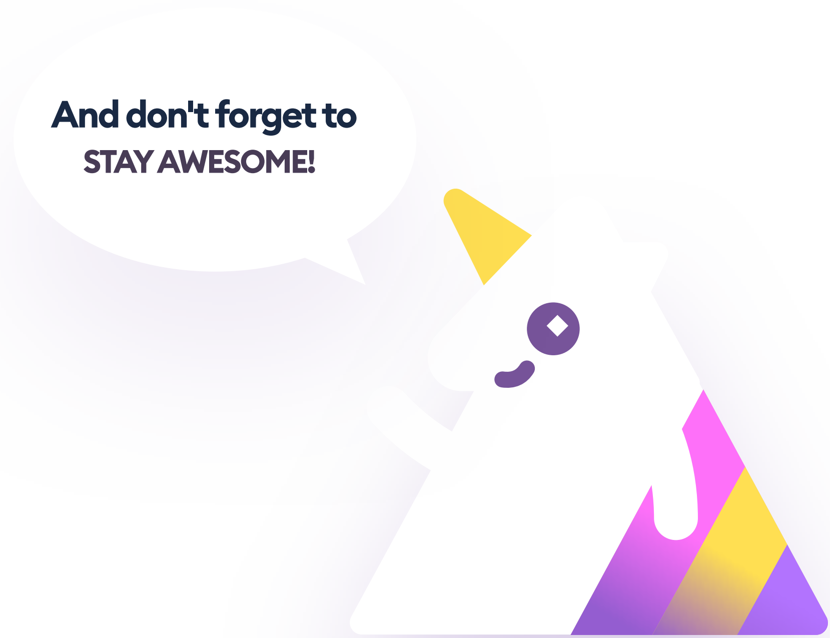
https://www.hype4.com

But why should you even consider becoming a unicorn? There are a couple of reasons, but the most important two are **money** and **satisfaction**.

By being versatile in improving your skills and also knowing enough of the surrounding skillsets you'll be able to get much higher hourly rates than people who "just code". Don't just code, be a unicorn!

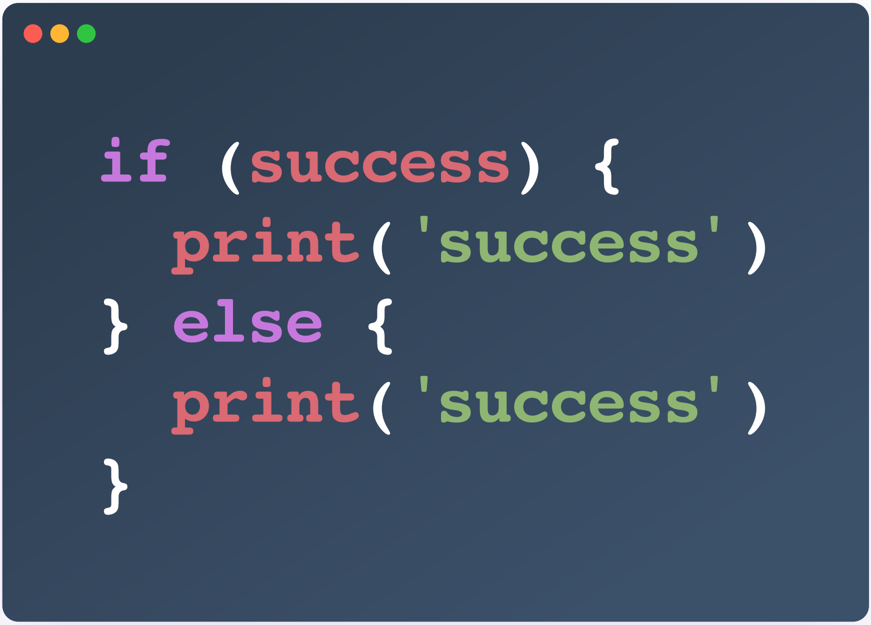
But money is one thing. After a while, being able to see how everything fits together and how you actually make a difference with your work is even more important. It'll help you to actually be happy with what you do, and this, my friend, is worth a lot more than money.

So strap in, let's begin our unicorn journey!



Joke time

We really found this in someone else's code ;) 1/7



```
if (success) {
   print('success')
} else {
   print('success')
}
```

1.

Who is this book for?

Who is This Book for?

LET'S START

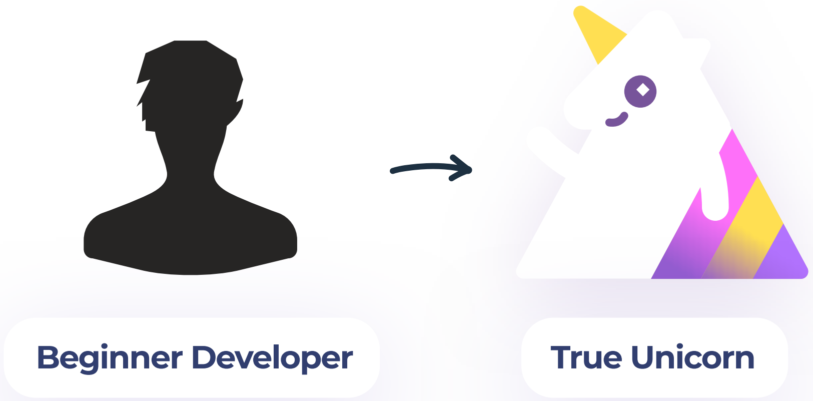
It's for front-end developers who treat their work seriously and want to grow as fast as possible.

It's not yet another coding tutorial. We will not teach you loops, conditions, and functions. You will not learn a hot new framework.

You can learn these things in many places already.

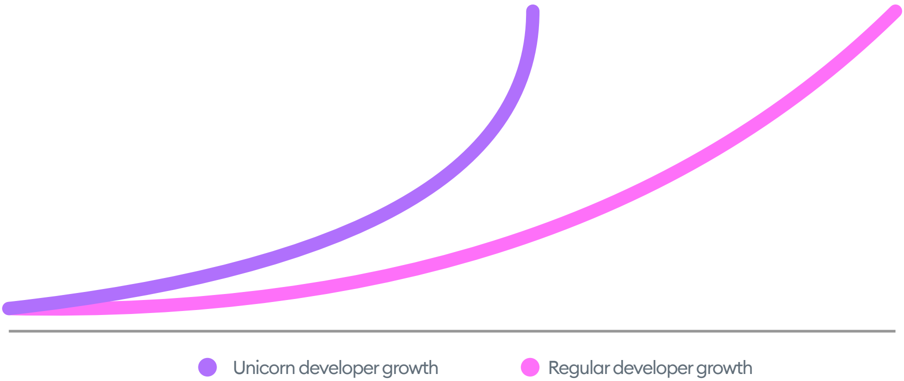
Instead, we'll show you how to transform from a beginner to an indispensable person in any team. We'll cover techniques, attitudes, and mindsets that will kickstart your career.

Coding is just a part of a programmer's job, often even not the most important part. To grow and get recognition, you need much more.



We'll show dozens of tips and tricks - you'll learn how to learn and communicate with others. How to use version control and work with designs. If it's the only way, we'll even help you become a designer! We'll guide you through the process of planning your career. You'll also learn some often overlooked front-end developer skills, like semantics and accessibility.

In short, we'll show you what you need to know to become a senior developer. That usually leads to a much higher hourly rates and a bigger feeling of accomplishment, so it's a win-win situation!



Your hourly rates can grow faster or slower. You can take the standard path and reach seniority in a couple of years, or become a unicorn and reach that point faster while also feeling happier about the quality of your work.

This book is not a silver bullet. A lot of work is waiting for you. The journey from someone who knows the basics of programming to a competent developer is hard and long. Our job is to make sure you understand what you need to learn and how to learn it. Your job is to put in the work.

Good luck, and have fun!

“We give you the skills and tricks, but you still need to put in the work.”

2.

How to find a job?

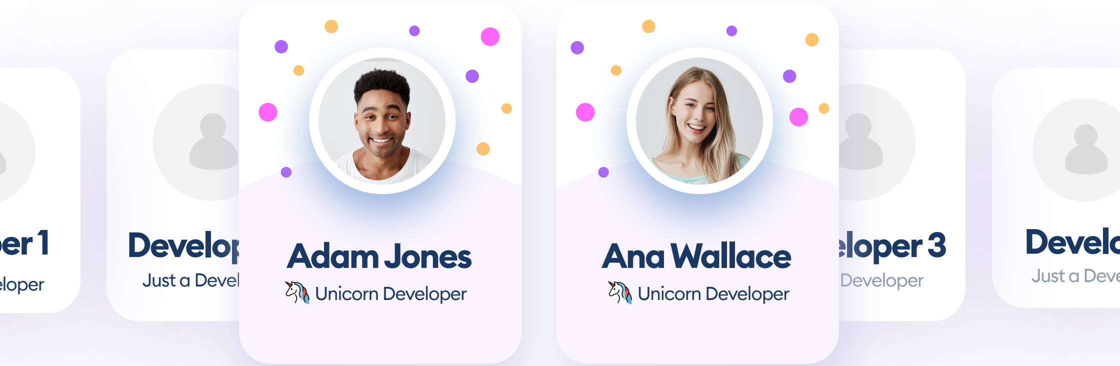
How to Find a Job?

THE BASICS

There are countless junior front-end developers out there, and all of them have a similar goal - to stand out. Whether you're looking for a job or want to become great at your current job you need to prove that you've got what it takes. This chapter will give you a high-level demonstrations how you can get there.

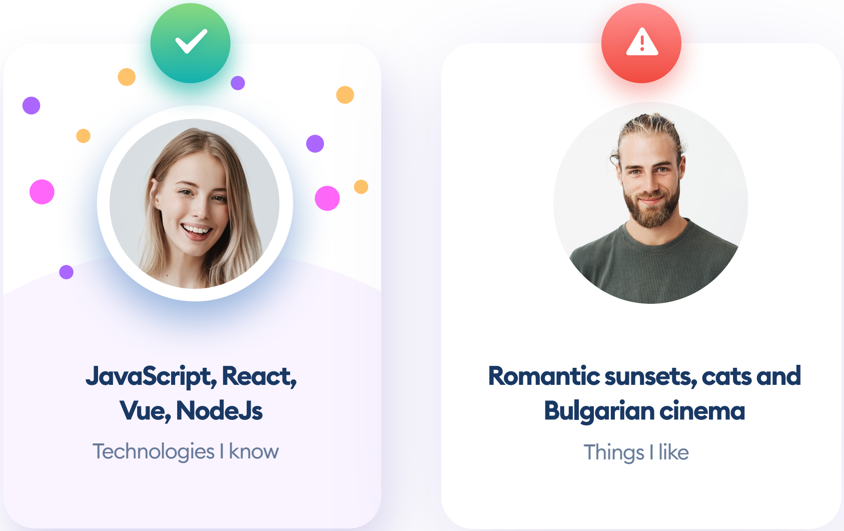
After all, front-end is the place to start your coding journey, so the competition is fierce. Nearly everyone is learning some form of front-end. There are tens or hundreds of applicants for a job. In all probability, more people will start to code, and the struggle to find a good job will become even harder.

But there is a silver lining. Most of the applications are indistinguishable. They're sent by people after these same courses, with identical projects in the portfolio. Even the CVs are alike, with an equal number of typos.
We'll help you to stand out in the sea of mediocrity.


er 1
eloper
Develop
Just a Devel
Adam Jones
Unicorn Developer
Ana Wallace
Unicorn Developer
eloper 3
Developer
Develo
Just a Deve

The first thing your employer will see is your CV and portfolio. Your goal is to make sure that's not the last thing he'll see.

Recruiters will devote less than a minute to assess your application. To get noticed you need to watch out for common errors and stand out.


JavaScript, React, Vue, NodeJs
Technologies I know
Romantic sunsets, cats and Bulgarian cinema
Things I like

So how to get through this first stage? You need to avoid typos, make it clear, and include only relevant information.

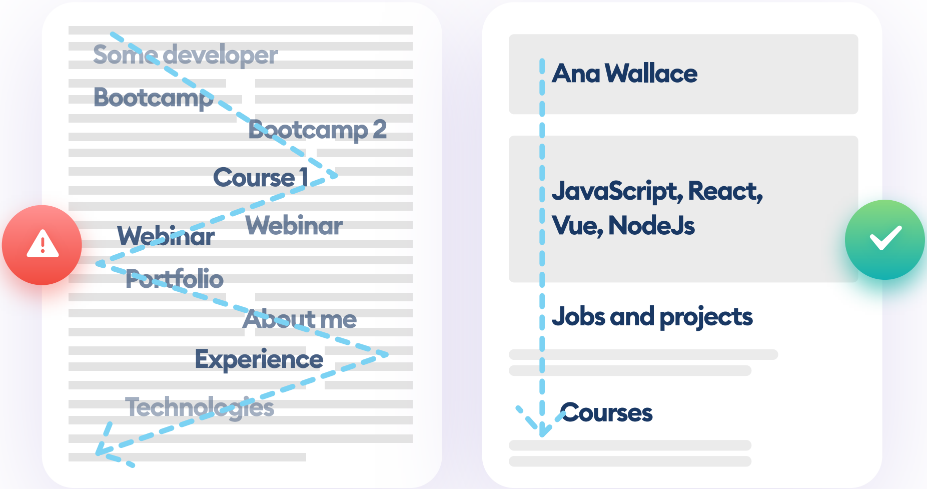
To make sure that you avoid typos and confounding sentences read your CV out loud.

You will feel the rhythm of the text, find chaotic sentences, and unnatural words. Resumes are going to be read by humans so you want to make them as easy to understand as possible.
If you can, ask a friend to read your CV. Not only will you get the second pair of eyes to find typos but also you'll make sure that someone other than you can still understand the message.

> “Recruiters will take less than a minute to asses your CV. Stand out and be clear.”

Go with a straightforward and clean design. Anything fancy will be just a distraction. That doesn't mean you can't use colors or different font sizes, but that you need to focus on readability.

Make sure that:
- the most relevant information is near the top
- different sections are clearly separated
- there's enough whitespace for it to be readable
- it has a flow from top to bottom, not diagonally or sideways


Some developer
Bootcamp
Bootcamp 2
Course 1
Webinar
Webinar
Portfolio
About me
Experience
Technologies
Ana Wallace
JavaScript, React,
Vue, NodeJs
Jobs and projects
Courses

The requirements for job offers are often a wishlist, and you don't need to meet them all. But remember to tailor your CV to the job offer. The recruiter doesn't need to know that you are a certified accountant or worked as a taxi driver five years ago. Include only the information and skills that can help you.

Also, it goes without saying that you don't lie. The truth always comes out and can ruin your future chances with other companies too.

Before you build your portfolio you need to choose your programming stack. That's a lot more important than you think. There are numerous libraries and frameworks used for front-end development and still more to come. You can't learn them all, and especially you can't pretend to know them all. The employer is not stupid. He knows that you started coding six months or a year ago, so it's impossible that you're an expert on React, Angular, and Vue, with some experience with Svelte.

If you are to focus only on a handful of frameworks and technologies, how to choose them?

You should start with the research on your options. Compare some popular choices like MERN (MongoDB, Express JS, React JS and Node JS), MEAN (MongoDB, Express JS, Angular JS, and Node JS), JAM (JavaScript, APIs, and Markup) and other you can find. Check what are they used for, follow a simple tutorial to examine if they feel right.

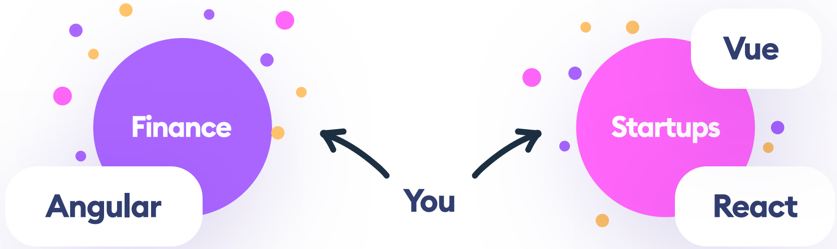
React
Angular
Vue
JavaScript
NodeJS
Adam Jones
Unicorn Developer
TypeScript

“

Choose your programming stack carefully. ”

With that basic knowledge, check job offers in your area. Notice how many offers there are, what types of companies are hiring, which positions (junior, regular, senior) are open.

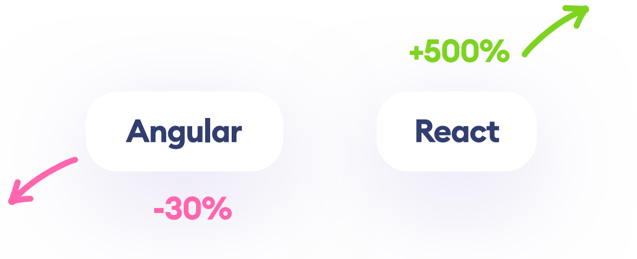
You'll observe some trends, for example, right now Angular is much more popular in the finance industry than other front-end frameworks, whereas React is widely used by startups. So the question is whether you want the financial stability of fintech, or the excitement and quick pivots of startups?



Another thing that you should take into account is the applicability of your knowledge in other areas. Some technologies create their own unique patterns of programming and ways to solve problems. Others leverage common features. You should favor more typical approaches.

During your career technologies will change many times, the deeper is your general programming knowledge, the better are your chances to learn new things and find new jobs.

The final step is to check trends in the market. There may be many job offers for a particular technology, but if it's declining you have to take that into consideration. It's typical that getting the first job in IT takes longer than expected, so you have to be ready for a long march. If possible, try to specialize in technologies that are on the rise.



Now you are ready to pick your programming stack. You need to find a sweet spot between your areas of interest, job opportunities, general technology applicability, and market trends

Possible path

I like **Startups**

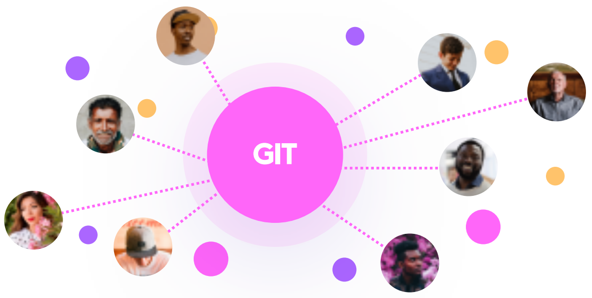
and **React** **is gaining popularity**

There's also a lot of **React** **job offers.**

The next question to answer is where to store your projects and how to show them.

You should absolutely use git for all of your code. It's like the center of the developer universe. It also has a number of other benefits.

It shows that you know how to use it, allows you to enjoy the benefits of version control, and is convenient to share with other developers. You can pick any service you like, but the most popular are **GitHub**, **BitBucket** and **GitLab**.



Additionally, you need a portfolio website. The recruiter won't have a time to analyze your code in the first stages of the recruitment. It's a lot easier to take a look at the website and test it a little.

Devote some time and make sure that your portfolio page works and showcases your work. Your website should resemble your CV - watch out for typos, stick to clean and clear design. You can add a few animations or other visual effects but nothing too fancy or disruptive.

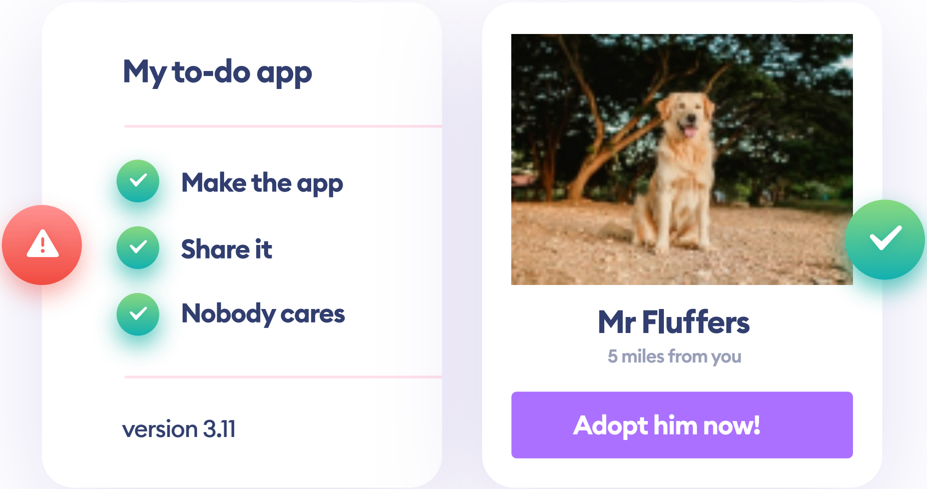
The focus should be on your projects, visual clutter will destroy the effect. It's also harder to maintain the good performance of the overloaded site, and some employers are assesing the performance of portfolio websites as well when making their decisions.

“

Knowing how to use GIT is essential. ”

Like almost every modern website, your portfolio should of course be fully responsive.

Don’t worry about the costs of hosting your portfolio. There are many free options like Heroku and Netlify and it's good to learn how to use them.


My to-do app
Make the app
Share it
Nobody cares
version 3.11
Mr Fluffers
5 miles from you
Adopt him now!

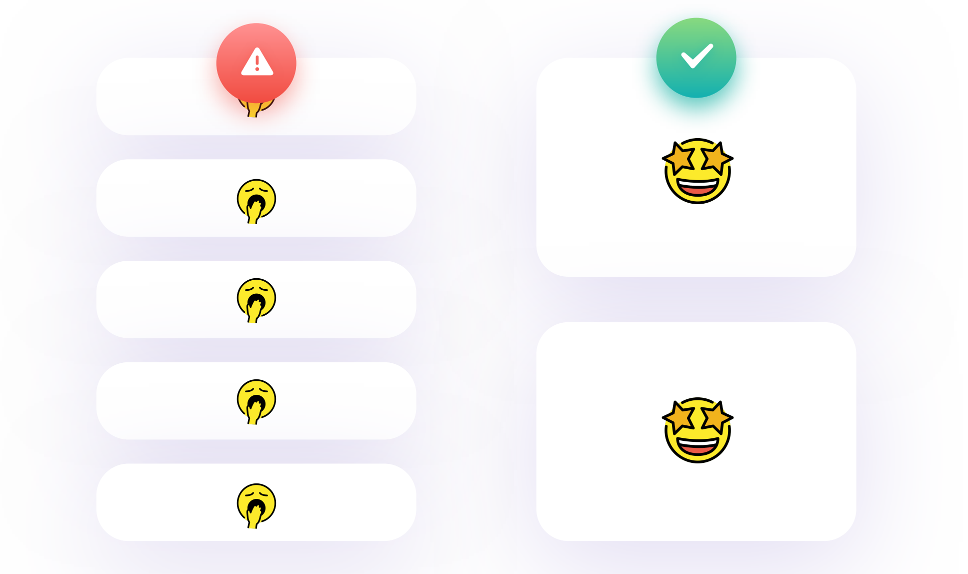
What to code?

It’s time to create projects for your portfolio. Avoid making another todo list or a weather app, you can do better than that. Try working on something that you genuinely like yourself. Something more original.

Do you like movies? Try creating a movie database like IMDB. Maybe you’re a Pokémon fan? Look for PokéAPI and make a Pokémon card game. Are you a dog lover? Make a site that helps to adopt animals using Adoptable Pet Data API. The possibilities are endless!

It's crucial that you find something interesting for you. Then you can search for free API's that will help you make your idea come true. Not only will you stand out amongst other candidates, but you also won't be discouraged easily working on the project that interests you.

The secret is in the quality. It's preferable to have two good projects than five copy-pasted, boring landing pages and apps. Show only things you are proud of and hide the rest.



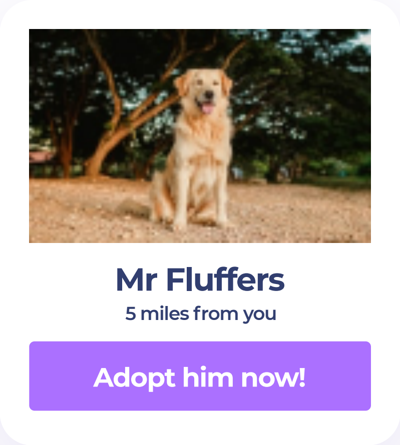
Focus on quality!

If you want to stand out, you need to understand what's around you. If you're looking at your job as a narrow, crowded path, you'll likely be stuck on it with hundreds of others.

But the goal of what you do - the goal of coding front-end - is to transform a vision into reality. So what is that vision?

The vision for a product starts with business decisions. Try to understand why the product exists. How is it going to benefit the company building it?

We will cover business analysis in-depth in future chapters, but for now here's a quick exercise:
Pick one project from your portfolio, and write a back-story for it. It's likely just a tutorial, but that doesn't matter. Imagine and name the company behind the product you've built.



EXAMPLE

Adopt a Pet!

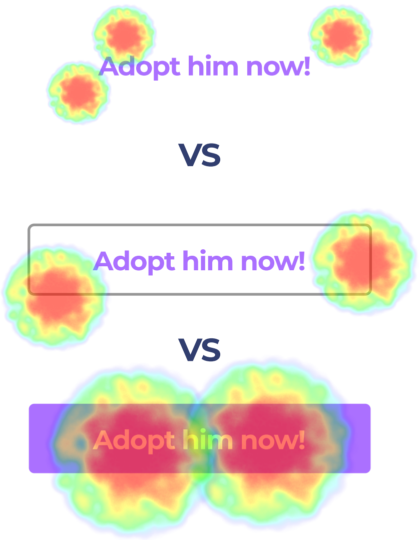
Adopt a Pet is a Startup that connects nearby animal shelters with people looking for their new best friends. It uses geo-location and saved preferences to only show when relevant animals show in the database!

Write down like the portfolio piece you've just created is a real project. Try to think of similar, but real products and see what their business model is. How do they make money? How user acquisition works? Is the goal to first get as many users as possible, or is it to earn as much as possible from the start?

Of course you don't need a PhD in economics or english lit. It can be very basic. But try and write just one paragraph of the back-story of the product. The more you do it, the more it will allow you to understand how things work around what you're doing.

After this part is done, the next stage is the User Experience. Understanding how UX works will make it a lot easier to work with designers in the future and good collaboration is one of the biggest assets a developer can bring to the table.

Install an app like hotjar on your portfolio piece (even the free plan) and send it to some friends to test it. Once enough of them click through, you'll have heatmaps showing how far people scroll (on average) and what exactly they interact with. You can then attach those heatmaps with a short description to your portfolio - it will definitely make them stand out.



EXAMPLE

Click analysis

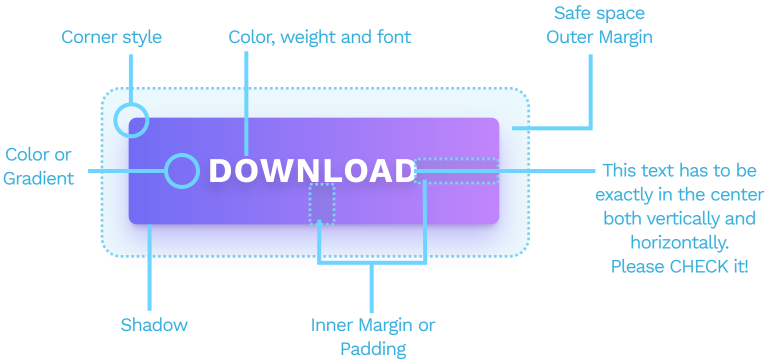
Quick HotJar analysis showed that a more substantial button makes the call to action stand out more and achieves a lot more resulting clicks.

Conducting research - and click testing is a form of user research - means that you also understand how to implement research tools into what you're building. This is going to be invaluable to the employer, or to your startup. There will be more about how to understand User Experience and collaborate with the UX people in future chapters.

> “Understanding the areas around yours will make it easier to collaborate.”

User Interface, or UI for short, is what you'll be working on the most as a front-end developer. It's also an area which most developers don't take that seriously.

There is an old developer saying, that if it displays on the screen, it's fine. That's the wrong approach. Try to consider HOW it displays on the screen. We will cover all the good UI rules in another chapter, but here's an example of one of the major problems in interfaces. Button text alignment.


Corner style
Color, weight and font
Safe space
Outer Margin
Color or
Gradient
DOWNLOAD
This text has to be
exactly in the center
both vertically and
horizontally.
Please CHECK it!
Shadow
Inner Margin or
Padding

You can use tools like the pixel-perfect browser plugin, to overlay the interface on top of the browser window. That will allow you to adjust your coded outcome, to fit the design with great precision.

Obviously design tools render fonts a little differently, than browsers do, so there's no need for 100% accuracy here. But as long as you're over 90% you should be safe.

The main takeaway from coding UI is, that the decision the designer made are not random. Or at least we trust them not to be. That means you need to recreate the design exactly as it is, with the same colors, margins, paddings and fonts.

Key takeaways

- ✓ To find a job you have to choose your tech stack. Don't be the jack of all trades, become a specialist
- ✓ If you're not a designer make your portfolio clean and readable, no need to be fancy
- ✓ Always prepare live versions of your applications - recruiters don't have time to go through the code
- ✓ One great project is better than 10 mediocre ones. Show only what you're proud of

3.

Stack decisions

Stack Decisions

DECIDING REQUIRES EXPERIENCE

Knowing what tools to use for a given job is often separating beginner developers from the more experienced developers. This alone is a significant building block of what we call "experience" regarding the developer's practice.

Seasoned developers are good in what they do because they had a chance to encounter more problems. They had to put the effort to understand them and find a solution. You could almost say that developers' seniority is measured by how many errors they've seen.

Wisdom earned during that process makes them prepared for other occurrences of the same issue in the future. The second time around, they will probably still have to look for a solution. But now they came prepared, with a few more tools under their belt, so it's going to get slightly easier each time.

Deciding on what tools to use is a crucial part of every project. It defines how you are going to approach implementing features and what your environment will look like. Developers who are just starting out most often do not have enough knowledge to leverage all the pros and cons of a framework or a library before using it at least one time.

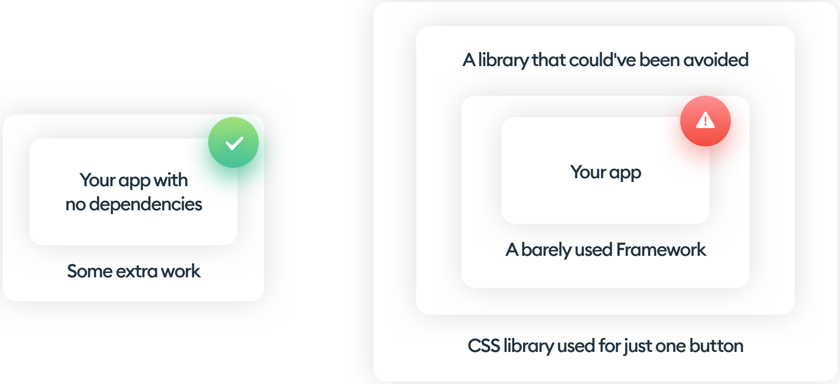
How could you make a decision then, without wasting time on something that is inevitably not going to work for your problem?

We have a few tips for that.

> “Deciding on what tools to use is a crucial part of every project.”

Before you start browsing GitHub or StackOverflow looking for the perfect library, you should know why you want to use it in the first place. What is the real problem you are trying to solve? Maybe you don't need a whole CSS library to style only a few buttons, divs, and some text; an entire HTTP library to make a few request call to one API server? Maybe what you already have is good enough.

Asking yourself those questions is an excellent way to prevent you from packing a project with hundreds or thousands of never used lines of code in the form of dependencies. You may think that it doesn't sound like a problem for you, but every dependency makes your app vulnerable not only to your own bugs but also to bugs in all the third party libraries. And you can't control that, apart from uninstalling the library.


Your app with
no dependencies
Some extra work
A library that could've been avoided
Your app
A barely used Framework
CSS library used for just one button

The other problem with that it increases the weight of your app. It's particularly important when it comes to websites, which should load as fast as possible. Every extra bunch of code that you need to import to your page makes it load slower. And that makes a significant impact on the User Experience.

Solving a problem on your own is what you should consider your first move when dealing with a coding issue. Use what you already know and try building it yourself. You'll feel much better about it when it works. Don't be this one developer that installs a package only to check if the number is even. The feeling of accomplishment alone is worth it.

My App
Useless Framework
other Framework
Huge CSS library

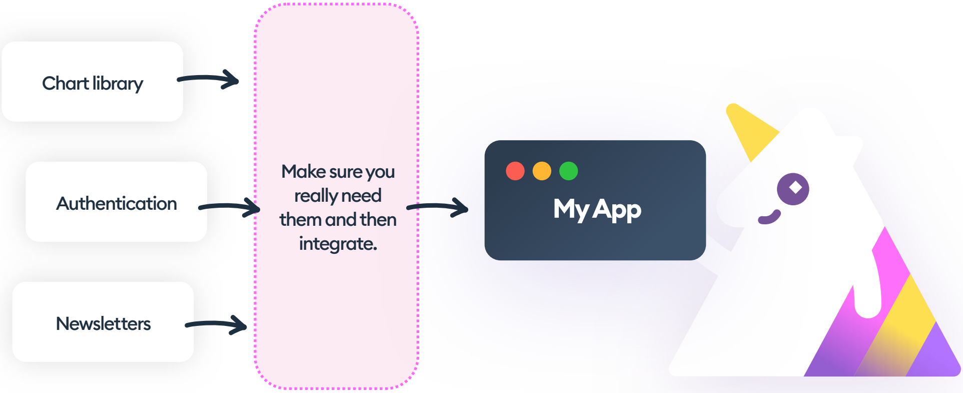
If you can't come up with a solution after a bit of time, you can try searching the web for a hint. But just a hint. What I mean is, that you should not go to StackOverflow immediately and grab the snippet from the first answer with a green checkmark, trying to make it fit into your code. For the most part, when looking for help, take the time to read the explanation.

You can find out if it's really related to your problem, or maybe it only looks like it should help. See the reasoning of the person that solved it. It's vital to your growth to understand the issue you are facing. Next time you'll know better how to look for it at least.

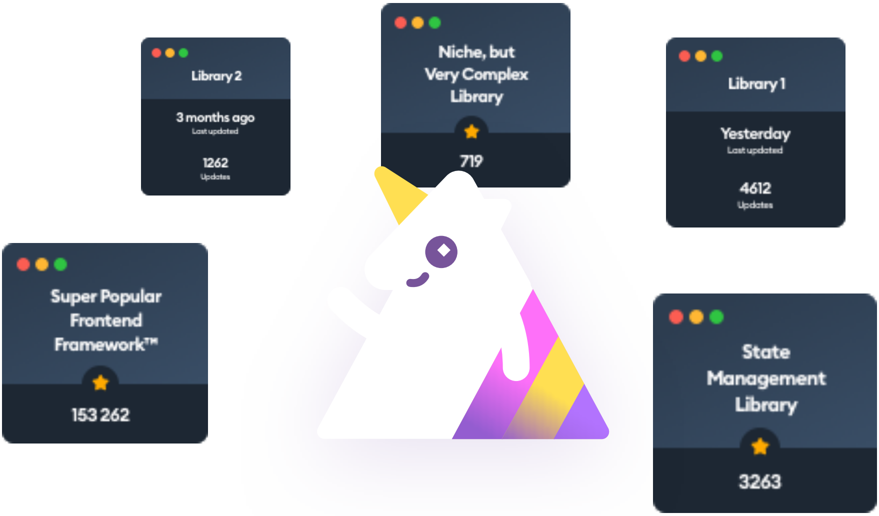
Here comes an important disclaimer. Most of the tips mentioned before apply to smaller parts of your code. Things that you can write in a few hours or a couple of days. They are meant to help you solve your bugs or implement simple features with greater understanding and care for your code's quality.

If your problem is related to a more significant part of the functionality and seems like a common thing that is not specific to your project only, there is a great chance that someone already encountered the same thing. It could be something like sending emails, authenticating your users, drawing charts, or building a whole layer of network connectivity. With features of this order, it's often useful to look at what others have already created.

Using a battle-tested library that's been in use for quite some time is a significant advantage over a custom code that you have to maintain yourself. It doesn't mean you should not stick to the principles we've been talking about before. You still have to make sure that you actually need this package or library. You should evaluate what impact it would have on the rest of your codebase. For example, check if it's going to make the page loading slower, etc.



When you come to the point when it's inevitable to use a third-party library, you need to decide which one you will use. How do you do that?
Often, complex features can be implemented in multiple ways, depending on the creator's design approach, code style, or experience.

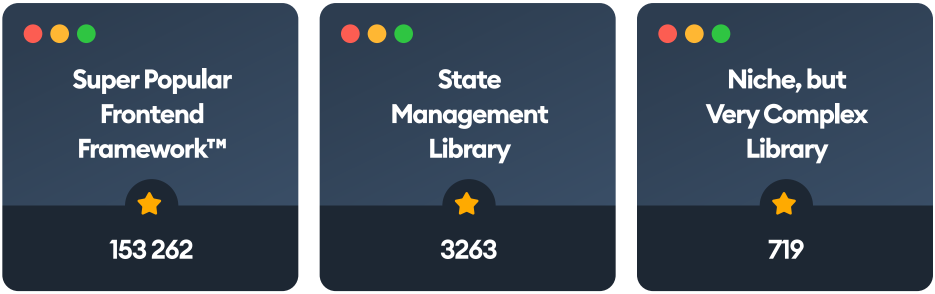

Library 2
3 months ago
Last updated
1262
Updates
Niche, but Very Complex Library
719
Library 1
Yesterday
Last updated
4612
Updates
Super Popular Frontend Framework™
153 262
State Management Library
3263

Which library should I choose?

You need to know how to pick the one that will be the best fit for you.
You might not know that the first time you choose a solution to a particular problem, but with time, you may come to use many of them and knowing which one to apply this time.

How can you minimize the error margin for the first time?

Let's look at a couple potential guides to picking the best one.


Super Popular Frontend Framework™
153 262
State Management Library
3263
Niche, but Very Complex Library
719

Different categories also vary in popularity. But just because something is not as popular doesn't mean it's bad. It may as well be something not many projects need.

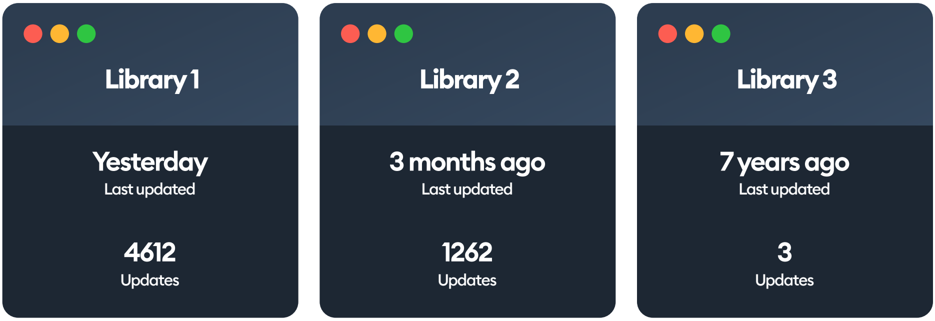
GitHub gives you many tools to help you assess the worth of a library. The number of "Stars" is probably one of the first things that many developers look at when doing so. It's a great way to see how popular the repo is within the community, but the number itself means nothing when you don't have a reference point. You need to know what number is a threshold below which you probably shouldn't trust a library.

And it differs between each programming language or the abstraction level of the library. For example, a popular frontend framework can reach well above hundreds of thousands of stars, but a state management library would probably rank lower than that within a range of a couple of thousands to tens of thousands of stars.

A specialized library in a niche technology should never be disregarded due to the small number of stars. Keep in mind that stars are a good measure of a library's credibility, but you need to know how to look at that number with perspective.

> “Number of stars shouldn't be the only factor you judge a library by.”

RECENT ACTIVITY



Library 1
Yesterday
Last updated
4612
Updates

Library 2
3 months ago
Last updated
1262
Updates

Library 3
7 years ago
Last updated
3
Updates

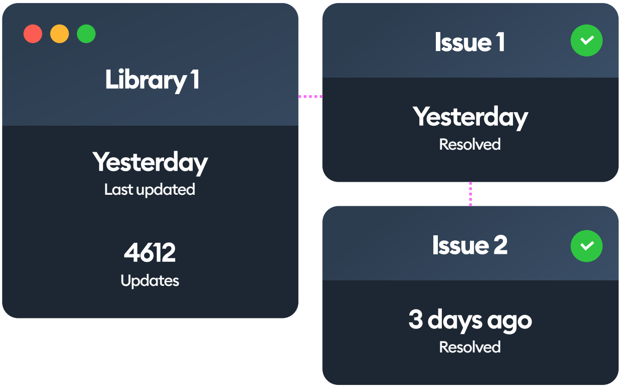
What other signs of a healthy library are there to help you if you are not sure after looking at the stars counter? The next thing you should consider is how active are contributors of the project.

See how long ago it has been since the last commit, if the library is updated regularly, how frequent releases are. Those metrics can help you make sure that you will not be left with an unsupported library anytime soon.

Here comes another disclaimer. When looking at any recent activity, it is worth considering if it's even required. It depends on the technology and the problem that the library is solving. It may be the case that the issue has been solved correctly and does not require any changes to the solution or frequent maintenance updates.

That's rarely the case, but it's worth noting. Frontend projects, due to the environment's fast-paced nature, often have to keep up with the changing landscape, so you should mostly stick to the actively developed projects.

ISSUES TAB



If the project looks like it's not actively maintained, consider looking at the "Issues" tab of the repository. Not only are you going to see if the maintainers are still resolving raised issues, but you can usually find helpful information there. Sometimes when projects get abandoned, people responsible for it don't even bother updating the description in the main "readme" file.

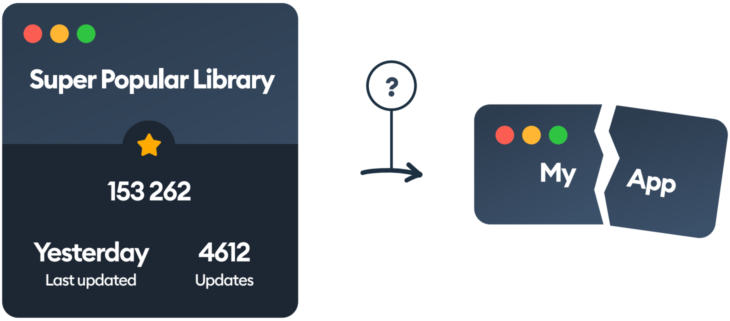
Finding a perfect solution, only to discover that it's not maintained anymore, is like a little heartbreak. If the idea is still valid and seems like it could work for others, the "Issues" tab is sometimes a place where you can find a thread about starting a "fork" of that repo by people willing to take it over.

The "Issues" tab gives you one more additional tool that can help you with evaluating a library. You can search through all the issues using keywords related to the technologies you already use. With this method, you can see if other developers encountered any major problems when trying to use them together. If that's the case, how did they solve them? Maybe it's not going to work for you either.

Diving deep into the code of a third party library was something I was hesitant to do initially. I thought that I would not understand the code because of the complexity and my programming skills being much worse than those who wrote it. I was completely wrong.

Reading the code that is applying good design patterns is going to help you in numerous ways. If the project you are looking at has a high star rating and is commonly used, you are going to read some quality code. And that's always a big win. It's not only going to help you understand this particular library's inner workings but also help you with getting around these kinds of projects and their structure.

Looking at the source is also a great way to make sure the library is working the way you expect it to and if it's actually going to help you solve your problem the way you want. It may be the case that after reading it, you will get inspired to write your own solution with a slightly modified approach that is going to be a better fit for you. At that moment, it's good to ask yourself if you and your team will be able to maintain such a module in the future. If that's the case, you are good to go.

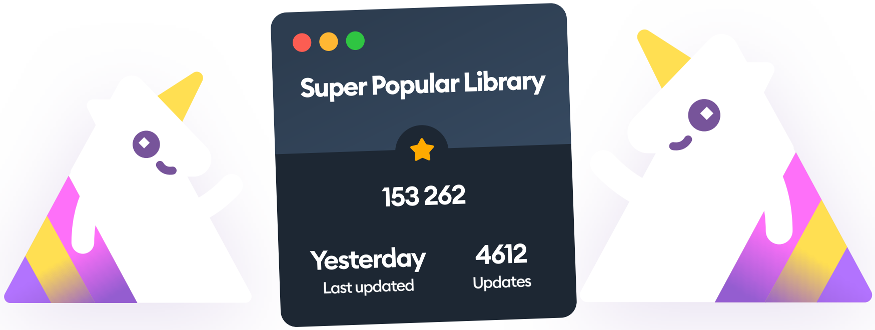


Say you found a third party library that you want to use in your app. It has got many stars, contributors are active, it looks like they are working on issues that users open, and the code seems good. What's left before using a package manager to install it? - Making sure that it's going to fit in your project.

When you add a dependency like that, you have to check how it will work with your current codebase. Is it going to integrate with it seemingly, or you will have to do a complete overhaul? Look at the programming style you are using. Do you need to change it to use the library? Maybe you are using Object Oriented Programming or Functional Programming, and the package is written with Reactive Programming in mind.

It's even more significant when you think about at what stage you are with the project. If it's in an advanced state already and you have to change many modules you wrote, it's probably not a good fit for you.

> “Make sure a library you choose will integrate with your code well before using it.”



This is the ultimate tip when deciding on using a library.

If you are not sure of a particular solution you found, go and ask other developers what they think. You do it all the time anyway by browsing Google and scrolling through StackOverflow, looking for an answer to your coding questions.

This is the perfect topic to bring up around developers you work with on a daily basis. Let them know what problem you are trying to solve, what you've already found, and how you evaluated the solution.

Then ask about their opinion. Maybe some of your co-workers had a chance to use this package and can share their experience. They should take an even more active part in the process if it's something that may impact their work as well. A simple package that you will use to generate some unique strings isn't going to affect your team's workflow the same way a state management library or a data layer client would.

At that point, you should have a better picture of how you can approach decision-making in the digital product development process. Tips we shared here can be applied to any other level of technology stack decisions. The bigger the role of the certain element, the more points to consider, and the more people should be involved in the process.

When it comes to picking a framework or a language for a project, you should not decide yourself without discussing it with your team, even if you are a senior developer who is leading the project.

The only exception is if you are a sole developer. You have to consider your team's strengths, their knowledge, and current experience in any other scenario unless you want to do all the work by yourself and let them watch you do it.

Key takeaways

- Try to keep the size of your web apps as small as possible by not using too many third-party libraries

- f you think you need a library to solve your problem, think again - maybe it'll be easier to implement it yourself

- Don't reinvent the wheel. If your feature is a common thing, there is a great chance that there is a well tested, open-source solution available

- Don't rely on GitHub stars as the only measure of library's quality - look at the open issues and recent activity

- Read the code of the library you want to integrate into your project to understand better how it works

- When in doubt - ask your team or other developers

4.

Understand design

Understand Design

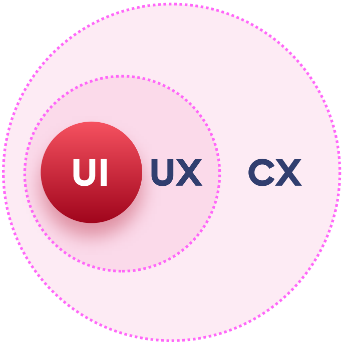
THE BASICS

These chapters in a more expanded form are also available in Designing User Interfaces. We tried to modify them to be easier to grasp by a developer, but there was no need to reinvent the wheel sometimes.

As a front-end developer, one of your main tasks will be implementing designs. Think of yourself as the builder, following an architect guideline. That's the standard way of understanding that relation.

Let's forget that for a second, shall we?

I believe, that you need to be both the builder and the architect to be truly great. That doesn't mean you need to **do the design** yourself (you can, however!). You need to be able to understand design completely and know the logic and reasoning behind the decisions. If you know the "why" of a design, you'll be able to better execute the "how" and achieve much better results. When we consider digital products, the main design language that comes to mind is User Experience, Customer Experience and the like. So looking from the top-level down let's see what they are.



Customer experience is a high-level category with UX being a part of it, while UI being a part of the UX process. The diagram shows the relation between these elements. As a developer you'll mostly be working with the UI part, but the outcome of what you do will heavily influence both UX and CX.

> If you know the "why" of a design, you'll be able to better execute the "how"

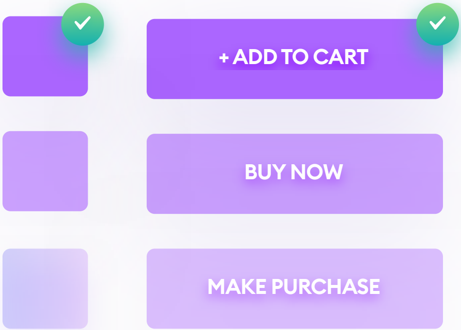
CX or **Customer Experience** is often confused as being another name for UX. The truth is that CX is a top-level process that defines not only how a product works, but how the entire company operates.

The CX process is the UX of the product, but also the branding (logo, colors and the like), marketing, or even how friendly customer support on the hotline is.

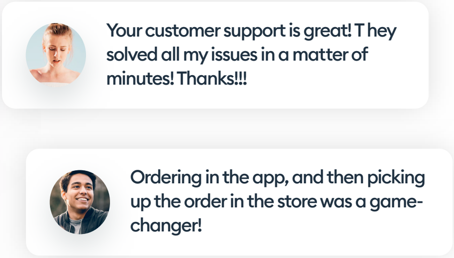
Customer Experience is about defining clear, consistent rules for the entire brand. Those rules apply to all the real-world experiences, as well as UX and UI. They even influence the way you write your copy.

A good example is one of the popular fitness apps. They changed the boring "save your workout" message to an "I'm awesome" button. Customer engagement went up.

All of those elements combine for a consistent, coherent vision with which our potential customer can quickly identify.



Changing the brand colors to a more positive, green gradient yielded an improvement in user perception of our brand. We also tested various types of label text to see what works best.



Your customer support is great! T hey solved all my issues in a matter of minutes! Thanks!!!

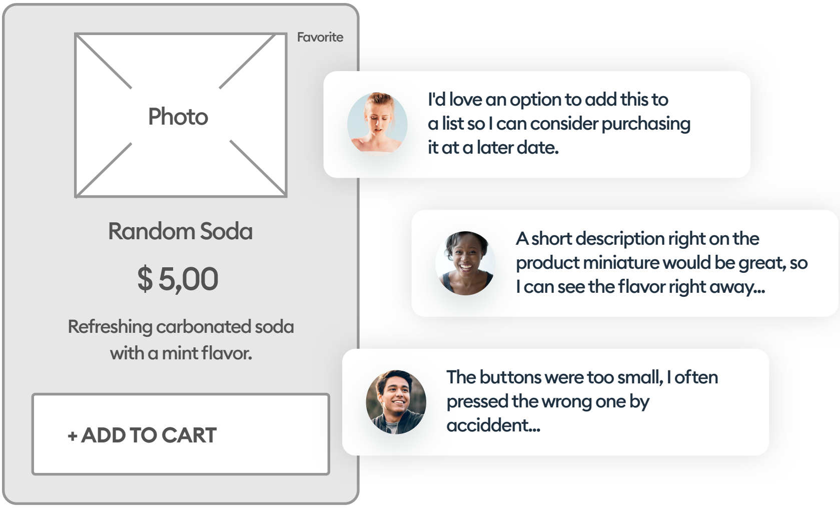
Ordering in the app, and then picking up the order in the store was a game-changer!

Merging the digital world ideas with the real world.

UX or **User Experience** defines and studies how easy it is to use the product. With digital products, that means the interface, navigation patterns, and communications. The goal of UX is to allow the largest possible group of users the ability to understand and use a product. If the role of a UI designer is the look & feel, the role of a UX designer is defining how it will work.

Aside from information architecture planning, UX also covers various kinds of research (surveys, A/B tests, focus groups, interviews, workshops, and more).

UI is a part of User Experience because readability and looks are also influencing the ease of use and shape our feelings towards a product. The way front-end is coded can also heavily influence the user experience - if for example objects are hard to read or hard to activate (by a click or a tap) it can completely break the experience.



I'd love an option to add this to a list so I can consider purchasing it at a later date.

A short description right on the product miniature would be great, so I can see the flavor right away...

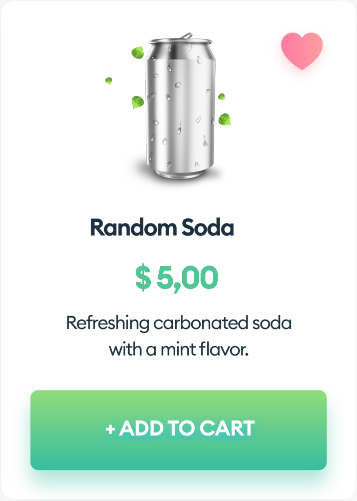
The buttons were too small, I often pressed the wrong one by accciddent...

A basic information architecture (wireframe) and user comments to implement.

UI or User Interface is the visual representation of a digital product. Its obvious use cases are apps and websites. UI is the link between the user and the functionality of a product. It helps to achieve the desired outcome through a series of human-machine interactions. It's a series of text, shapes, graphics, and photographs combined in a way that allows for a fluid, natural interaction.

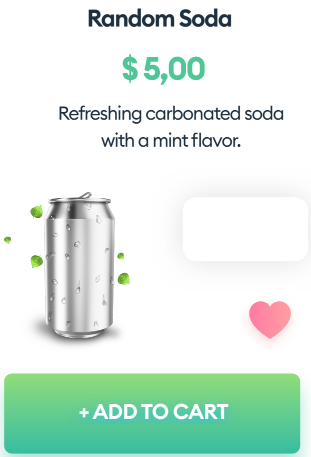
An interface is a mix of grid, layout, typography, colors, animations, and micro-interactions. In other words, UI is all that we experience - mostly with our eyes. UX defined information architecture is a blueprint for UI's refined final look.

A **UI designer** is responsible for the final look & feel of the product. The job is to define its unique style and make it fit the target market. That visual representation should be readable, usable, and devoid of any unnecessary, distracting elements.



Refreshing carbonated soda with a mint flavor.

Product Card



Refreshing carbonated soda with a mint flavor.

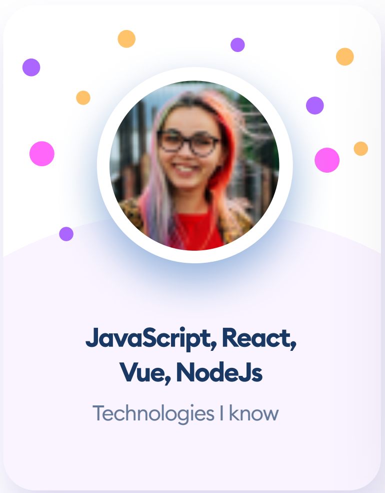
Component set

“As a developer you can make or break the user experience. ”

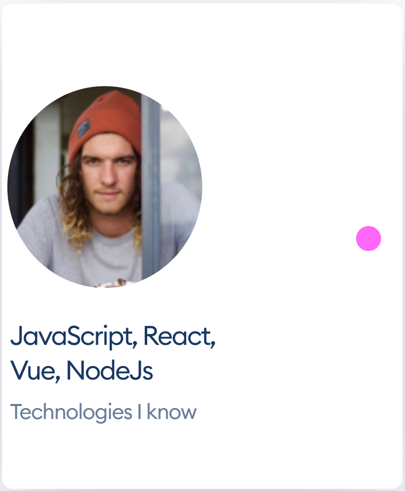
Let's assume you're working with the best designers out there. The way you implement their designs can either make or break the entire user experience.

Many front-end developers live by the rule of:
"If it displays on the screen so my work here is done"

This is wrong! The HOW here is just as important as IF. You can either recreate it "pixel-perfect", or create a Frankenstein monster that sort-of resembles the starting point, but not really.


JavaScript, React,
Vue, NodeJs
Technologies I know

The initial design representation.


JavaScript, React,
Vue, NodeJs
Technologies I know

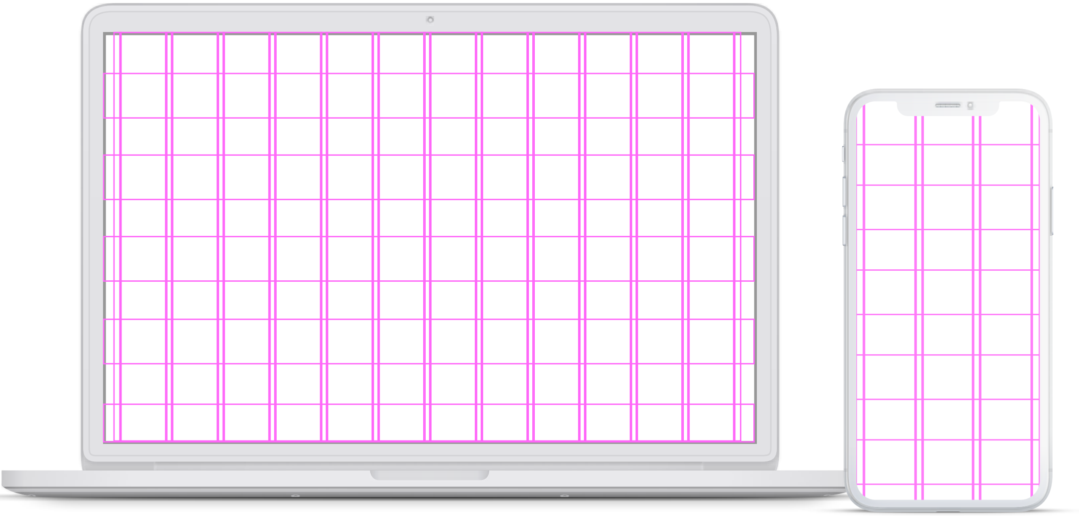
The "coded" version. As you can see almost every element is done the wrong way. All the design rules are broken here, but it does "display on the screen".

You don't want to be that kind of a developer! You can not only implement the designs perfectly, but also suggest animations, transitions or other creative ideas to make them shine even more. There are some basic rules to follow and we'll go through them on the next pages.

Grid

THE BASICS

The very first thing to grasp from the design world is also the most fundamental one. The grid is what can make a product more readable, or completely messy. Understanding it is also the very basics of Front-end development.

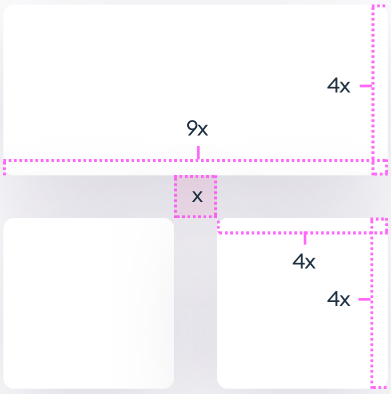


Grid is a structure of invisible lines that helps keep your layout together. It's a structural blueprint of any good project as it helps to create the hierarchy between elements and allows for better understanding and processing of on-screen information.

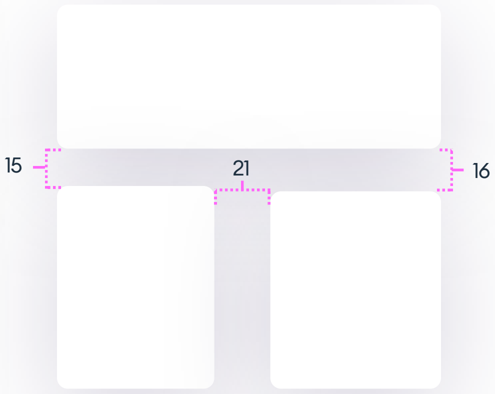
Or in other words it's what you align every object to.

Our brain scans information and looks for patterns it can quickly identify and merge into larger groups. That grouping process is essential for understanding complex elements.

This is where grid comes in. It's a series of alignment and sizing rules that helps with that processing, because our brain can follow the "invisible lines" a lot better, than when everything is completely random.



A very simple soft-grid based on the number 16 (x). This is the simplest and easiest way to have a grid that works.

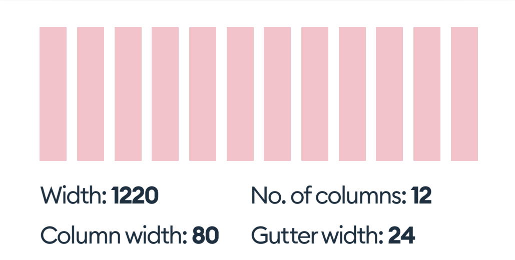


A completely random grid that makes the information much harder to process.

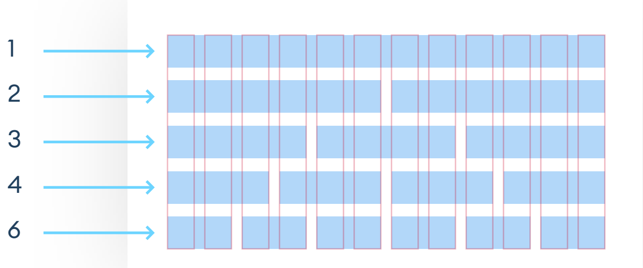
When developing a grid, make sure that all the values match. Look for inconsistencies and either fix them yourself, or ask. Designers are people too and they sometimes make mistakes. If most of the margins are 16 points, but then you see one that's 17 it's very likely that it's a mistake and you should simply use 16.

12 COLUMN WEB GRID

One of the safest web based grids is a 12-column layout based on a 1220 width. It caters to most popular laptop resolutions, while giving the most flexibility with the content.



It's also important to understand why we're using 12 columns. It allows you to create sections on the website in the most possible configurations (1,2,3,4,6 possible equal section widths)



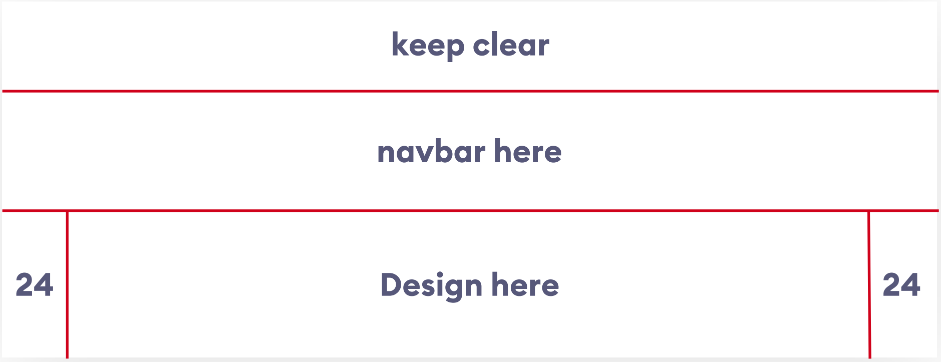
Of course in most cases the grid is already defined by the designer, but if it isn't then a 12-column one is the best place to start with. Sometimes the designer forgets about the grid and creates a design that doesn't follow any precise guidelines. In cases like that try to convince them that they should modify it to match a 12-column grid, as it's going to make your job a lot easier as well.

> “A 12-column grid is one of the most versatile settings for desktop web design.”

MARGINS ON MOBILE

Mobile grids can be intimidating. If you're building a mobile app, the best approach is to make a soft-grid starting with setting the side-margins. These are “lines you don't cross” on both sides of the content. They’re also something a lot of designers don’t pay too much attention to.

When the left and right margins are even slightly different, it shifts your entire design optically to one side and has a negative impact on readability.



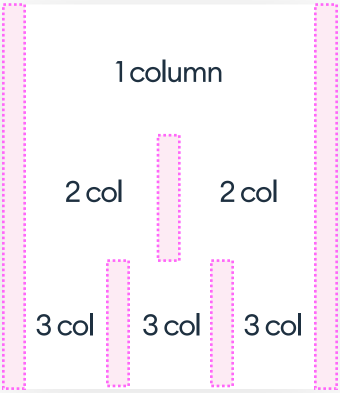
The main two soft-grids right now are an 8-multiple and a 10-multiple.   It means that 8 and 10 respectively, are usually the smallest margins your content can have. Then 16 and 20 are one step higher, with 24 and 30 being the next set of values.

Design needs to breathe, so for most mobile projects I set the margins at either 24 or 32 points. For dense projects with a lot of data in column views, it’s best to go with 24, and for simpler, less condensed screens let's use 32. It's a safe way to end up with a nice looking result.

Ok, but why shouldn't we go lower than 24? After all going with 10 or 16 will allow us to fit more content onto the screen, right? Yes, but more content doesn't necessarily mean a better outcome. If the content is too close to the edge (and both 16 and 10 are a bit too close) our brain needs to also process the relation to content with the side of the screen line. That slows down our processing and makes the content basically harder to read.

This is also why most books have pretty wide margins - it greatly enhances readability.

If you set your margins at a specific value - let's say 24, you can use the same value to then build the soft grid of main on-screen components.

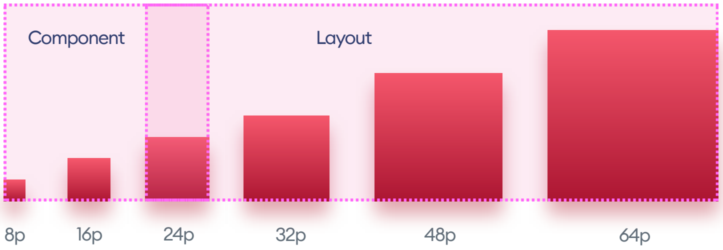


Soft-grid means that you only work with the gutter widths - or margins - and keep the columns equally divided in the remaining space. when we use our side margin to achieve that we can end up with one, two or three possible columns. Making more than three columns on mobile phones is usually not a good idea, as the columns would be to narrow to be readable.

SPACING TO DEFINE HIERARCHY

The general rule by which our brain processes visual information is through spacing-defined hierarchy. It means that we tend to naturally group objects that are closer together, and detach objects that are further apart. That is helpful if you want to be able to quickly scan and understand a complex visual set.

Knowing that, make sure you create at least 4 spacing values so the resulting layout is readable.

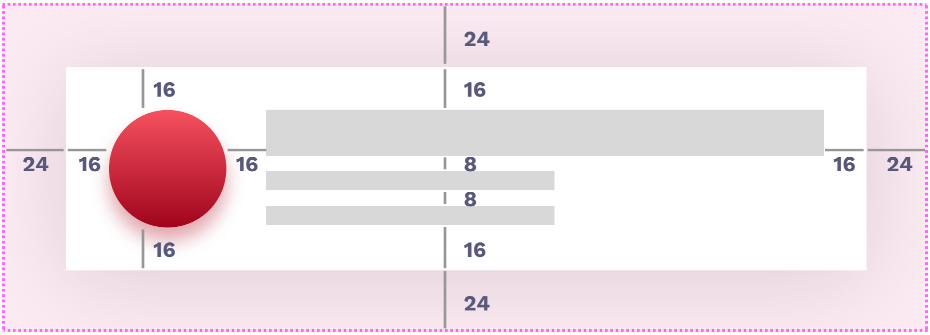


Objects (like buttons, cards, action sheets) are grouped into components, and those components are displayed on an overall screen layout. Above, you can see an example of an 8 point grid.

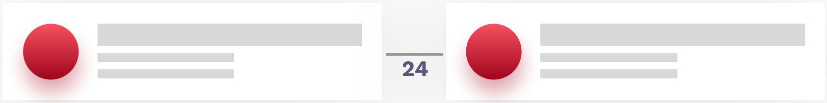
It's important for hierarchy, to create a clear distinction on what's inside a component. For that you can use smaller spacing values for inside the component (like aligning objects within a card) and larger values between components that form a layout.

Sometimes there is an overlapping value that can belong to either a component or the general layout (in our case 24) but that clear gradation will be extremely helpful when defining a useful interface grid.

The closer the objects are, the more they are perceived as a group that belongs together. When designing a card layout, obviously the content inside the card will be closer together, than the individual cards themselves. The main benefit of blockframing is that it's a lot easier to apply that rule. Let's look at it from a perspective of one single piece of the layout - the card.



The card here is the main "group" so it's positioned 24 points from each side of the screen. The circular photo is another group, but it resides inside the card, so it's positioned at 16 points from each side. The text content blockframes are yet another group, so they're positioned 16 points from each side, but the space between them is 8, to show they belong together. Use that rule to easily group common sections together.



So steppig back from the main card, we can the use the larger number (in this case 24) between the cards for better hierarchy.

> Our brains love to follow a precise, consistent grid. Make sure that's what you build.”

Key takeaways

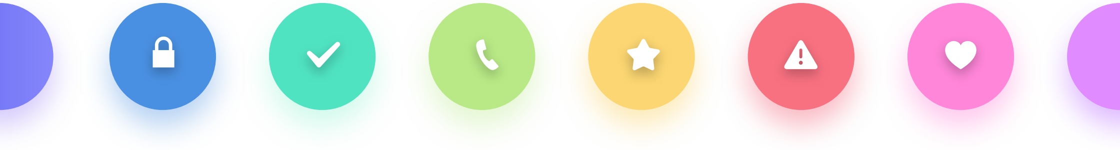
- ✓ Keep your grid precise and consistent!
- ✓ Spacing defines hierarchy. The closer objects are, the more they are perceived to be a group.
- ✓ Soft grids use margins and paddings to define the column width based on the device screen size. They are great for mobile views.
- ✓ 3 columns are likely the maximum column density for a mobile screen.
- ✓ 12-column grid is the most versatile for desktop web design.

And if you understand grids, this is exactly where the next item should be!

Colors

THE BASICS

The next thing after the grid that you need to be aware of is colors. In this chapter I will share the fundamentals you need to understand to be able to build a color palette that won't hurt people's eyes.



Over 90% of whether we like a product or not comes from its color palette. To pick the right hues, try to establish the target market, message, and style of your product down to the emotions you want your future users to have when looking at the screen.

People remember colors more than almost anything they see, so doing this part right will either make your product memorable, or make it disappear among many similar ones. Often, people who don't remember the product name refer to it as "that blue app" or "that black and white website."

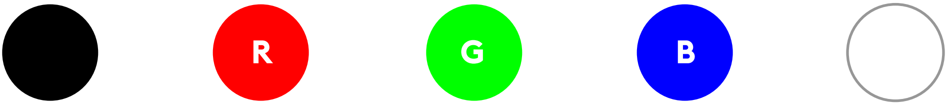
Avoid the defaults!

The easiest thing to do, could be to use one of the presets you can find in the front-end framework of choice, but that leads to the problem of the product looking exactly the same as many others. That can lead to the product having a lot less impact. But to successfuly modify the default palettes, you need to understand how colors work. Let's dive in!

There are a couple of color formats that you'll use as a developer. Some are more exclusive to working with designers than others however, so it's good to have an understanding of them.

RGB

It stands for Red, Green, Blue and is the most common way to specify colors in code. You can use the values from 0 to 255 (or 0 to FF in Hex) to represent each color.



0,0,0 or #000000

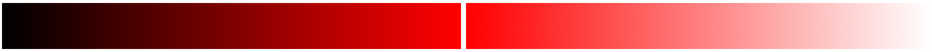
255,0,0 or #FF0000

0,255,0 or #00FF00

0,0,255 or #0000FF

255,255,255 or #FFFFFF

Getting lighter versions of each color simply requires to adding the exact same amount of the two other colors to it. In case of Red at 255, the bigger the value of Blue and Green, the lighter the red color appears. The numbers keep growing all the way until they reach full white at 255,255,255 or FFFFFF. And if we go the other way, it simply fades to completely black.



0,0,0 or #000000

127,0,0 or #7F0000

255,0,0 or #FF0000

255, 127, 127 or #FF7F7F

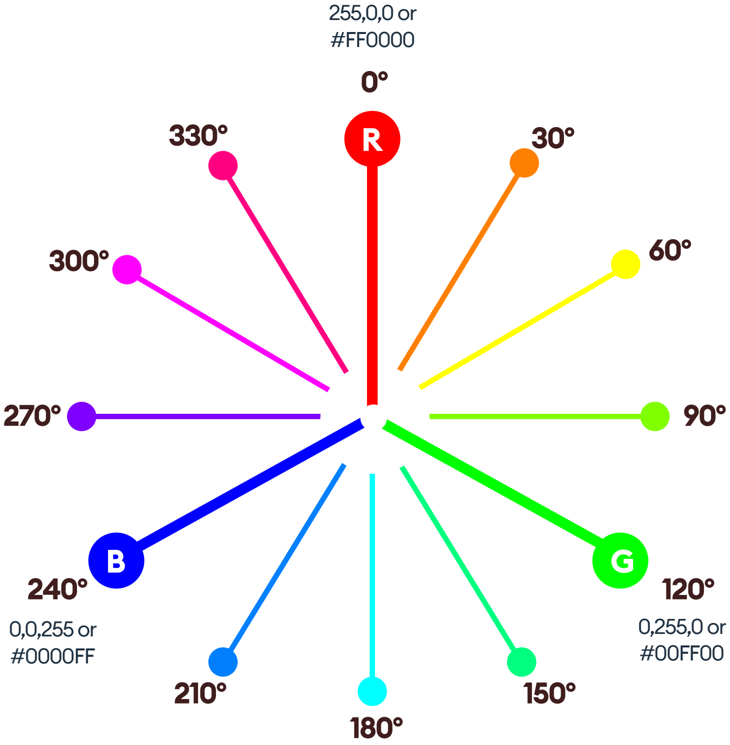
255, 255, 255 or #FFFFFF

RGBa

There's also a version of this format called RGBa, where the a stands for "alpha" or transparency. This value can also be a number from 0 to 1 (with decimals). So an alpha value of 0.5 simply means there's 50% transparency under that color.

HSB

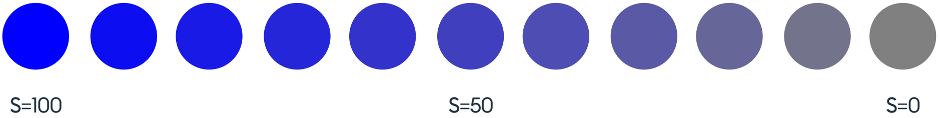
It stands for Hue, Saturation, Brightness. The first part - hue - defines the actual color as an angle on the color wheel. It starts at 0° which is Red, and finishes at 360° which is also red. The best way to show it is the diagram below. Sadly, in most design tools while switching to HSB you still get a rectangular color picker instead of a color wheel, which can make things confusing.


255,0,0 or
#FF0000
0°
R
330°
30°
300°
60°
270°
90°
B
240°
0,0,255 or
#0000FF
G
120°
0,255,0 or
#00FF00
210°
180°
150°

Notice, how 0°, 120° and 240° also represent the full RGB spectrum, while other hue values represent all the colors in between.

Saturation

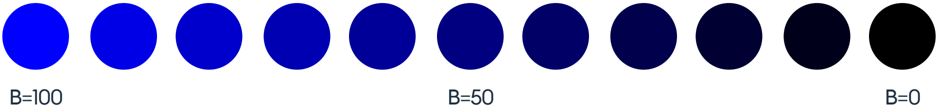
Saturation is a value between 0 and 100 that determines how saturated our selected hue is. The lower the saturation, the more grey the color will look like.



The example above is the same hue of 240 (Blue) with Saturation ranging from 100 to 0.

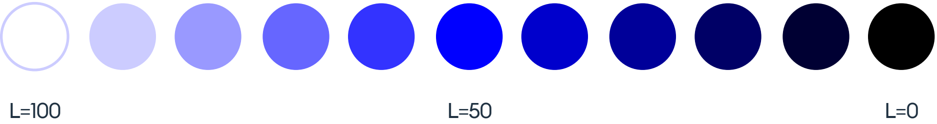
Brightness

Brightness is another value between 0 and 100 that determines how bright our selected hue is. The lowest brightness results in full black (#000000), however the full, 100 brightness is not white, but rather the full color Hue - in our case 240 (Blue, or #0000FF).



HSB vs HSL

There's also a color format that uses Luminance instead of Brightness. The main difference here is that 100 Luminance will actually give us full white, so our main color of 240° hue (Blue) is half-way through Luminance.

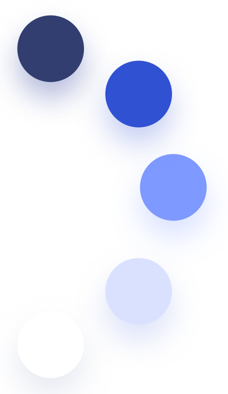


What is a color palette?

A color palette is a set of colors that work well together, combined to form a brand or a concept. It helps to set a mood for the final design. A pallette can use colors close to each other on the color wheel or include accents from the opposite sides. A typical color palette should contain both neutral and accent colors. It's also good to include notification colors (green for success and red for failure) - especially if forms are an essential part of the functionality.

You can find many color palette generators online that can help you choose the right hues and shades that match your initial color idea. Before you start using auto-generated sets, however, it's worth knowing how they work.

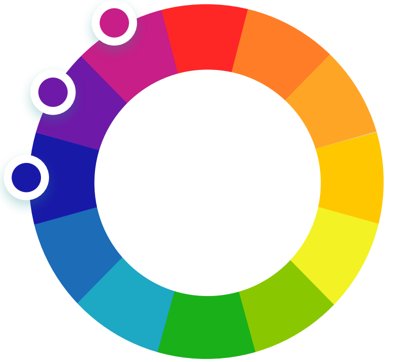
Here are the most popular color palettes out there, with a brief explanation each.



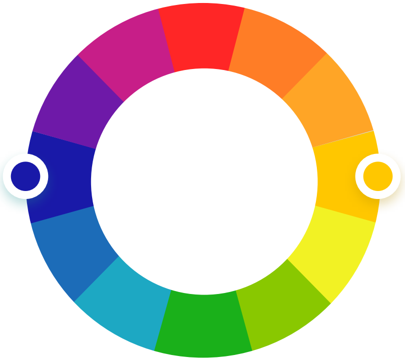
A monochromatic palette uses the shades of just one color as the entire set.

It allows for using almost the entire spectrum of that shade, including black and white. Monochromatic is a safe choice, as it's challenging to come up with clashing colors in a set like this.

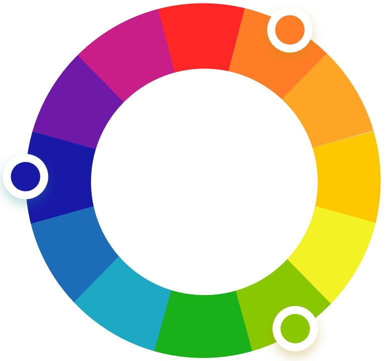
The downside is the contrast can be a bit lower than with other palette types, and our designs can end up looking quite dull.



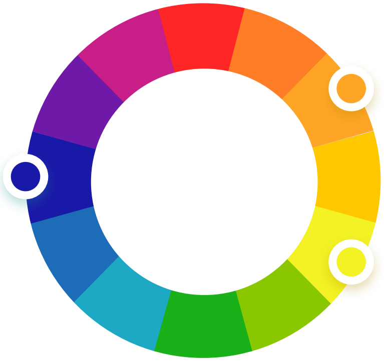
The analogous palette is a collection of colors that sit closely together on the color wheel. The downside of this choice is a slightly lower contrast between the colors, than in other palettes. To make it work, try experimenting with the shades.



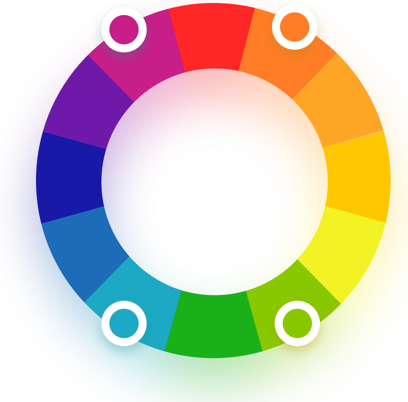
The complementary palette uses colors that sit on opposite sides of the color wheel. This way helps with achieving a high contrast between the hues. The best way to find the matching set is by choosing colors with less than 85% saturation.



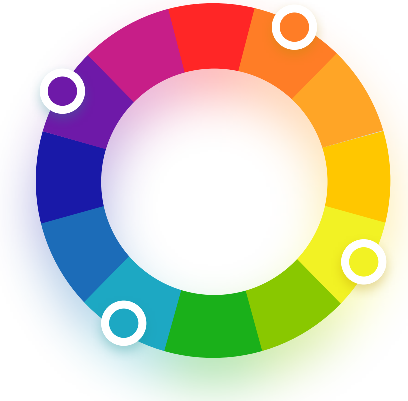
The triadic palette is an equilateral triangle on the color wheel. You can start by choosing its tip, and then pick the resulting two bottom corners. Triadic color sets are often more colorful and playful. When using this palette, try to set your primary color as the tip of the triangle, with the other two being complementary.



The split-complementary palette starts with choosing one color and adding two colors on both sides of its complementary hue. It forms a narrow triangle on the color wheel. This type of palette is a safe way of achieving excellent results that work well together.



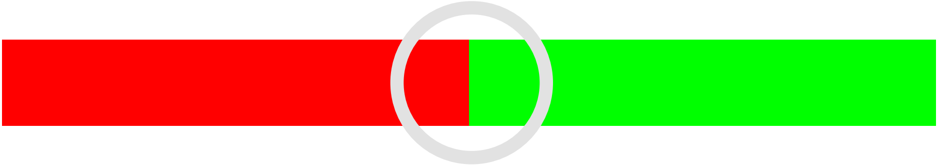
A rectangular color palette uses four colors positioned around the color wheel in the shape of a rectangle. The best way is to choose one complementary set first, and then choose another one that forms a rectangle.



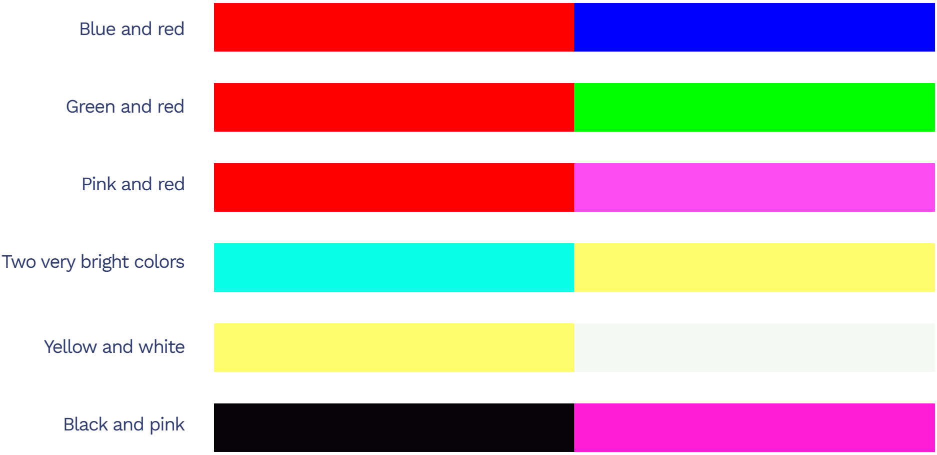
The square palette is a twist on the rectangular one, with the main difference being the two color spaces between our corners. The resulting shape is an even square.

COLORS THAT DON'T GO WELL TOGETHER

There are some color combinations that nearly never work well together. They're either an eye-sore, low legibility, or both, thus decreasing the overall quality of the product. Avoiding them is the best way to a great design, so you should always be aware of that. But to understand which combinations to avoid, you should always look at the edge between the two colors. If after a few seconds it starts to look fuzzy and unpleasant to look at, it means the colors you've chosen are clashing.

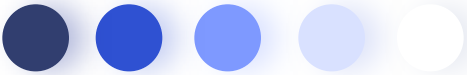


That's because our screens emit light right into our eyes and our focus jumps between those clashing colors without being able to discern which one is more important. That creates the "shaking line" effect. The colors below are a couple of example that clash and should be avoided.


Blue and red
Green and red
Pink and red
Two very bright colors
Yellow and white
Black and pink

> “Red doesn't work well with green and blue, especially with higher saturations.”

The best starting point to a palette is picking your base color. By just using it and shades of grey (with a hint of that color) you can end up with a very nice, monochromatic palette. You simply achieve that by playing around with the saturation and brightness and keeping the hue exactly the same.

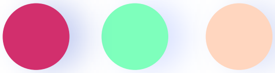


This is what you can get with using the hue 227 and playing around with brightness and saturation.

In most cases creating a lighter or darker version of our main color is simply about increasing or decreasing saturation and brightness respectively. The correlation here is pretty simple.

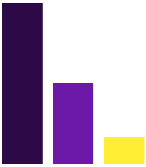
Darker color: Saturation is higher, Brightness is lower
Lighter color: Saturation is lower, Brightness is higher

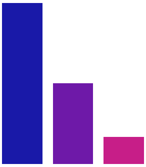
If you need extra visual clarity for complex systems, you can pick the safest "status" colors (success, neutral, failure) to compliment your basic, monochromatic palette. The most common colors for these states are green, orange and red. So to make them fit the above color scheme simply pick the hues that are the closest in saturation and overall style to the colors you already have.

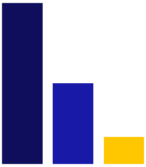


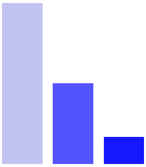
These three status colors are created with picking the red and green on the other sides of our blue, and picking an orange that's desaturated enough to fit the vibe.

When choosing a color palette, it's also essential to balance the colors in it. The best approach for a good palette balance is to use the Golden Ratio.

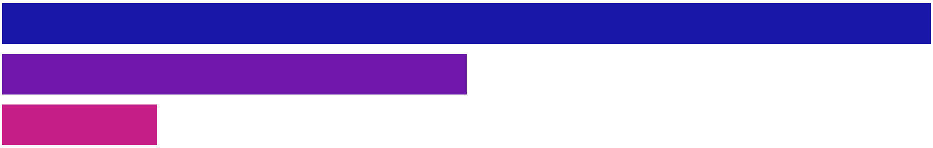








The Golden ratio rule is based in nature and has application with spacing and font-sizes as well. But for now the most important thing to remember is to simply use the ratio of 60, 30 and 10 respectively.



The example above is:
60% Dark Blue, 30% Purple, and 10% Pink.

Dark blue is the primary color - all of its shades (darker and lighter hues) are best suited for backgrounds and low-priority elements. It takes about 60% of the visual color space in the design.

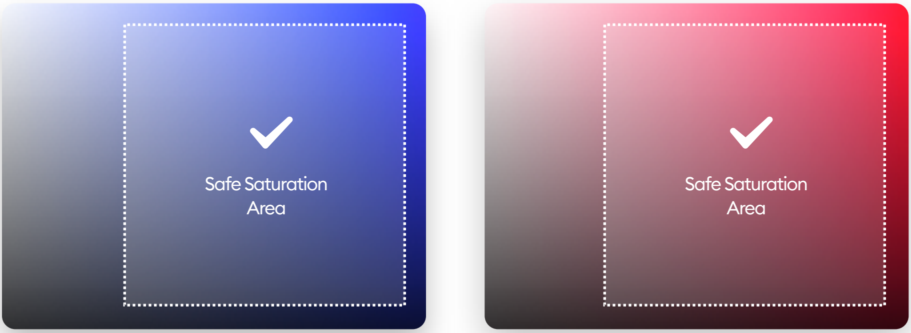
Purple is the supporting hue, great for higher hierarchy objects. For it to be sufficiently visible, its use is limited to about 30%.

Pink is the color for accents and CTA's, so 10% coverage works best, as it highlights the most important elements like buttons and selections.

Contrast is one of the most important metrics when it comes to end result readability. It can't be too high or too low.

In cases when contrast is too high on an already bright display, it can cause a strain on our eyes. It's good to find common ground between accessibility and aesthetics.

If you decide to use highly saturated, very contrasting colors, try to do that purposefully. It can work on event or product pages, but may not be the best choice for a news portal, e-store, or a banking app.



A good practice would be to avoid exceeding the 90% color saturation until you get comfortable enough to choose the right color combinations. The image above shows a typical safe area to pick safe-looking hues.

For your initial projects, or when you don't have a designer to help you, simply pick colors from that range for best results.

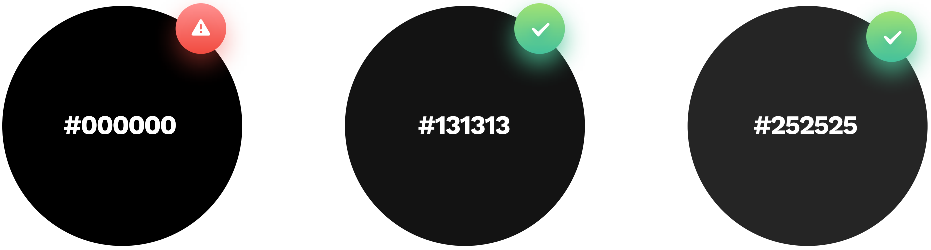
> Avoid using pure black (#000000), use dark grey instead.

PURE BLACK COLOR DOESN'T EXIST

Using pure black (#000000) in a digital product has been causing controversy for a few years now. Should you use it?

While pure black gives the best possible contrast (especially on a white background), it's not the most optimal choice for digital design. A contrast ratio that high can negatively affect the readability of the interface.

Ink on a sheet of paper is, of course, as black as it gets. But every time we look at it, the light bounces off of it differently. The result is often some shade of dark grey, as pure black doesn't exist in nature, which makes our eyes less adjusted to it. We perceive pure black as something unnatural.



Because the screen is glowing, it creates an effect of a "no brightness" color that glows (except for OLED screens), which can be very unpleasant to look at.

When we keep the contrast in the safe ranges, it's best to use a dark grey instead of pure black for nearly every possible scenario.

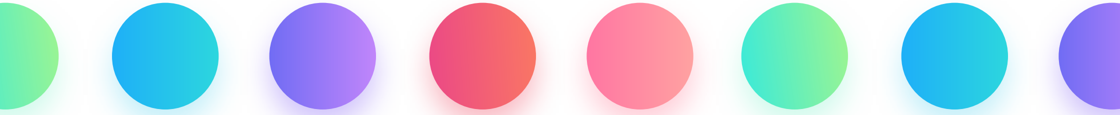
Key takeaways

- ✓ Avoid using pure black (#000000), use dark grey instead.
- ✓ Avoid colors saturated over 90%. They are much harder to work with and still achieve a great result.
- ✓ Clashing colors shouldn't be used together. The worst combinations are red and green or blue - having them close to each other creates a truly jarring effect.
- ✓ You can create a nice looking, monochromatic palette by playing with just the Saturation and Brightness.
- ✓ Use the golden ratio of 60/30/10 to balance your colors.

Gradients

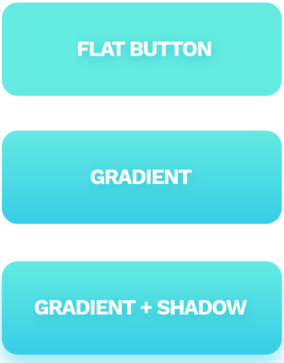
WHY GRADIENTS?

Gradients are more natural, than solid colors. That's because most of the colors we see in the real world have some form of color gradation on them. They look more natural, friendly and pleasant.



Because of their increasing popularity, many designers started calling gradients "the new colors."

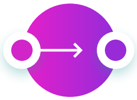
Gradients have to main purposes:
- One is the ability to create depth and a more defined shape.
- The other is about catching the eye and guiding it to certain parts of the shape. Our eyes prefer bright, warm hues, with pretty high saturation.

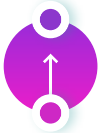


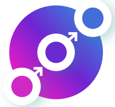
Even a subtle gradient will make your buttons more friendly and "clickable."

Depth allows gradient + shadow combinations to appear a bit closer to the eye - which helps in understanding the interactions behind them.

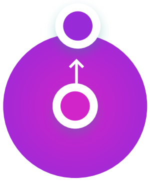
A gradient is a transition between two or more colors. The colors can also have be transparent. There are three main types of gradients:

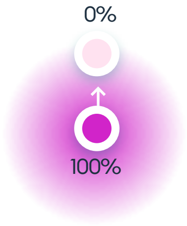


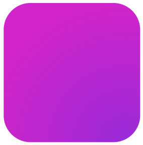




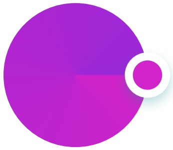
Linear is the most common gradient type. As its name states, it's a linear transition of two or more colors. We can modify the colors themselves, their transparency, and the gradient angle. It can be horizontal, vertical or diagonal.

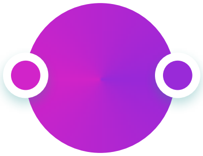






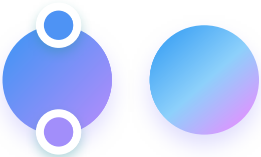
A **radial** gradient starts with one color in the middle of the gradient ellipse, while the other ends up being on its edge. The most common type of this gradient is the circular one, where the transition is distributed evenly along the circle. You can, however also use it on rectangular shapes or triangles to achieve a more natural effect than with a linear gradient.





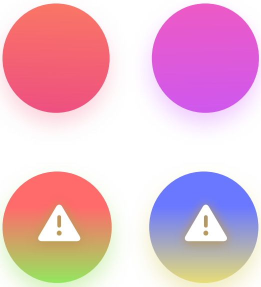
An **angular** gradient goes around in a counter-clockwise circle. The angle between the colors defines whether the transition goes both ways (180-degree angle) or has one sharp dividing line in the middle (at 0-degrees). In most cases you should simply avoid this gradient.

How to create a gradient that both looks good, and makes sense for great interface design? There are a few basic rules to follow to achieve that.



TWO COLORS

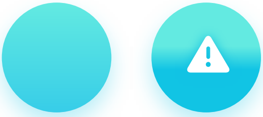
Two-color gradients are the best choice for most designs. With more colors, the gradient can become visually busy.



CHOOSING THE RIGHT COLORS

Try to match similar colors, warm hues with warm ones, and cold with cold.

That will help you avoid bad mixes (with the most extreme one being red and green)

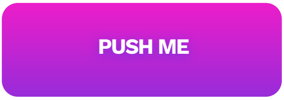


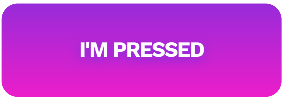
SUBTLE GRADIENTS

The best gradients are often very subtle, with not a lot of difference between the colors. It makes them look more natural and easier on the eyes.

> “Avoid gradients made from colors that are clashing or that are too far away from each other on the color wheel.”

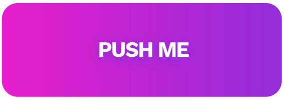
Gradients are a popular choice for buttons and other interactive elements. Here are some tips on how to make them shine.

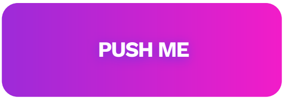




VERTICAL

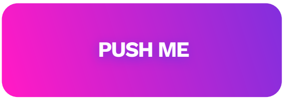
Vertical is the "classic" button gradient. If the lighter color is at the top, it looks more 3-dimensional and almost asking the user to activate it. If you reverse the colors, it can look like a pressed state.





HORIZONTAL

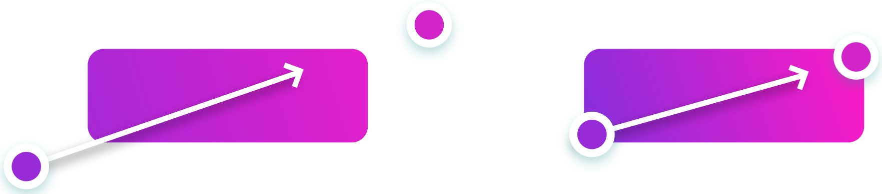
Horizontal gradients may not make as much sense in the 3d space, but they are gaining popularity because they look a little less generic than their vertical counterparts. We lose some of that convex style, but choosing the right colors can make the button look more dynamic and inviting to click.



DIAGONAL

Diagonal gradients help us achieve that organic, natural look. If you're trying them out for the first time, start safe, with a 45-degree angle.

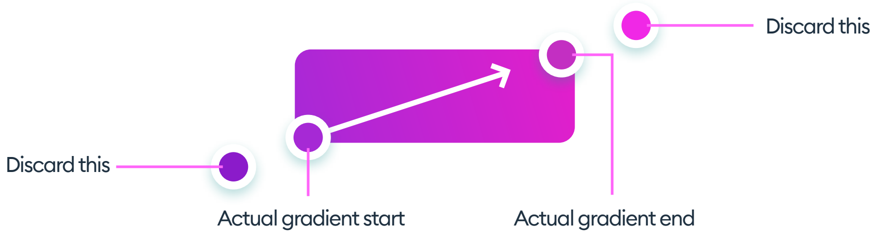
There's one common problem developers face when they get CSS values for soft-gradients from designers. This is due to the fact, that some designers, to make their gradients look more friendly, actually stretch the color handles way outside of the object.



```
background-image: linear-gradient(225deg, #FF1AC6 0%,
#842EDD 100%);
```

These two shapes have the exact same CSS properties, when copied from the design tool. The problem is, that the gradient handles are outside of the object in one, so using that CSS will zz result in the object below. As a developer you simply won't be able to recreate this effect. There are two ways out of this.

One is to ask your designer to modify the gradients so they're not out of the object bounds. The other is to simply use a color picker tool and pick the "real" gradient colors yourself. Then, you simply ignore the original colors to achieve the desired effect.



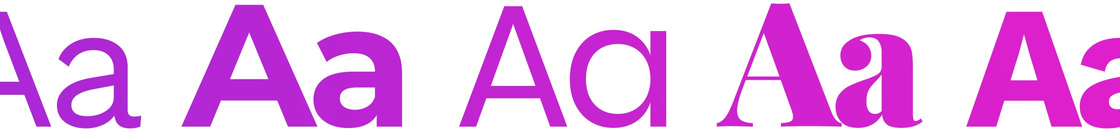
Key takeaways

- ✓ Gradients add visual depth and make the UI look more natural.
- ✓ Vertical gradients on buttons are a bit boring. Experiment with horizontal or diagonal.
- ✓ Linear gradients are the most popular. Radial gradients can look very nice on non-circular objects. Angular gradients are practically absent from UIser Interfaces.
- ✓ Subtle gradients made from two shades of the same color are the best place to start.
- ✓ Avoid gradients made from clashing colors (like green and red)

Typography

THE BASICS

Typography is one of the broader subjects in design, and there are many books dedicated solely to this matter. It's also the most difficult part of the design process to master. The fonts you choose profoundly influence the look & feel, style, and message of the entire project. Here's a developer-friendly overview on the best typography practices.


Aa Aa Aa Aa Aa

FONT OR TYPEFACE?

Some people call them fonts. Others call them typefaces. But what is it really? Who's right?

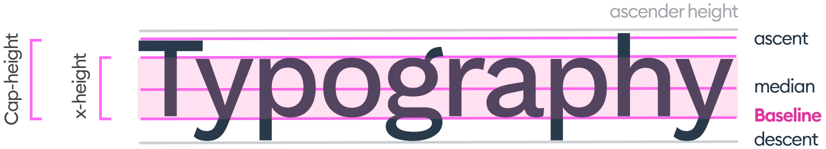
It's actually pretty simple:

A **Typeface** is a collection of various styles of the same lettering.
Each of those styles is a **font**. So Helvetica is a typeface, and Helvetica Bold is a font. Helvetica Regular is also a font.

A font is a digital representation of how the letters look. That presentation is about all the individual letters and their relation to each other.

FONT STRUCTURE

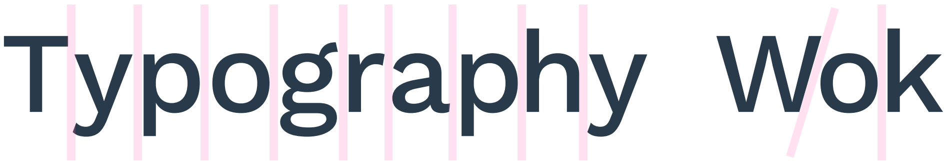
Each font has a set of characteristics, based on its structural definition. It's good to understand how precise and relevant these properties are.



There are many rules of alignment when it comes to fonts, but one of the most popular and easiest to implement is to align to the Cap-height. It means you use the capital letter as the bounding box for your font. It starts at the top of the letter, and ends at the baseline - which is the line on which the letters are "sitting". Descenders (ends of lowercase j,y,g,p and q) can safely be ignored when it comes to aligning fonts.

KERNING

Kerning is the space between letters. All fonts have a default kerning value that differs between them.

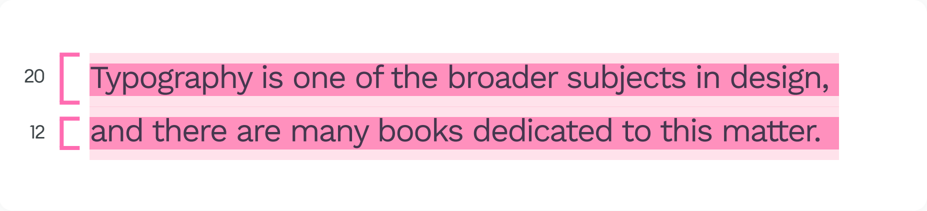


It's challenging to come up with many different kernings (letter-spacing) in one project. That's why you should convince your designer to stick to one or two at most. When the kerning is too small, or too large, it can negatively impact readability. If you're not sure how to set it, a safe range in which you can play around with the default value is between -0.5 and +0.5.

Leading is the vertical space between lines of text. Or as developers call it: line-height.

In mobile apps, it's usually a slightly lower value than on the desktop. The default value can often be too small for the readability of larger blocks of text. You can try to increase it slightly.

You can also use the golden ratio by multiplying the font size by 1.618 to get the best line-height value.


20
Typography is one of the broader subjects in design,
12
and there are many books dedicated to this matter.

In the above example, we have the text size set at 12 points. Next, we multiply this by 1.618 and get 19.4. After that, we round the number to the closest value and get 19.

The problem here, however, is that our text size is an even number, while the line-height is an odd one. It means it likely won't distribute evenly inside the line.

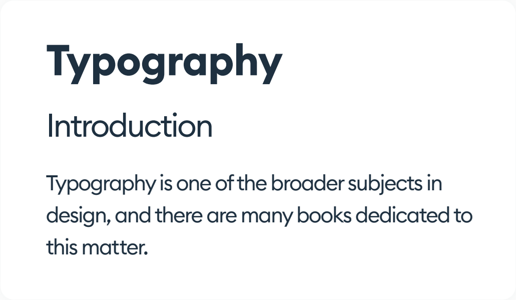
We round it up to 20, which is the next even number. The same rules apply to other font sizes. If your size is an odd number, the line-height has to be odd as well.

> “
>
> If you're not sure, set the default letter-spacing for the font, and use line-height between 1.2 and 1.4 em. ”

WHITESPACE

Whitespace is the empty area between letters, words, lines of text, and everything else we see on-screen.

Too small or too large whitespace value negatively impacts readability and hierarchy. Whitespace is essential to a good layout, and getting it right takes a lot of practice. Soft-grids can help you define whitespace between objects, but it's crucial to understand the typography values first.



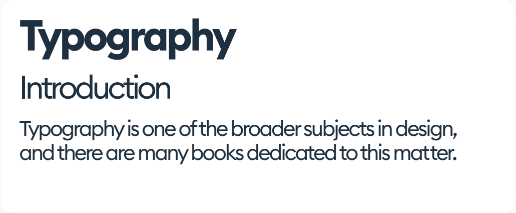
Typography

Introduction

Typography is one of the broader subjects in design, and there are many books dedicated to this matter.

SAFE

This example shows large enough margins and spacing for all the elements to be readable.



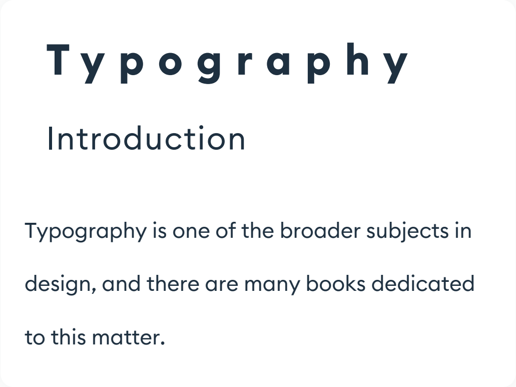
Typography

Introduction

Typography is one of the broader subjects in design, and there are many books dedicated to this matter.

TOO SMALL

When the kerning is a very low number, and everything is very close together, it's a lot harder to read and process.



Typography

Introduction

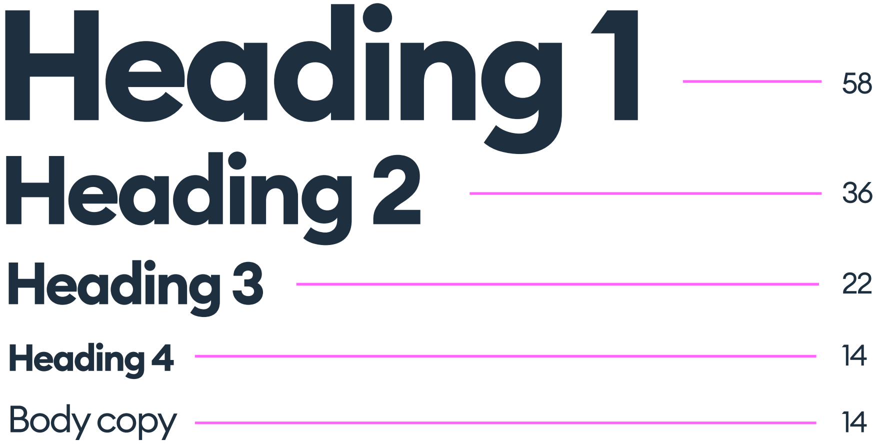
Typography is one of the broader subjects in design, and there are many books dedicated to this matter.

TOO LARGE

Uneven margins with very high whitespace make design look chaotic and hard to read. It seems as if there are no rules applied to the layout.

You can also use the golden-ratio rule to quickly come up with a matching set of body-copy and headers. The only thing you need is to come up with the initial value for body copy. A value between 14 and 16 points works best in most cases.

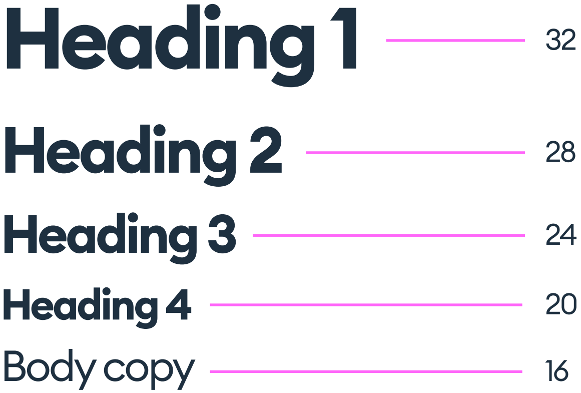
Then you multiply that value by 1.618 to create H3. Then multiply H3 by the golden ratio number to create H2. And all the way up to H1. H4 can be the exact same size as the body copy, just differentiate them with the font-weight.



Above you can see an example of defining the size of all of the headings using this rule. If you think H1 is too big, you can also move the sizes down by one value, so H3 and H4 would differentiate by font-weight from body copy, and H2 would be the size of H3, with H1 the size of H2.

This simple calculation brings hierarchy and order to the relation between the fonts and can help fix typography in any interface.

There's one more rule that's even easier to implement. If you decide on an 8-point grid for your layout, you can simply add 4 (because it's half of 8) to each consecutive font-size to create hierarchy. To better differentiate H1 you can add 8 to it instead of 4 too.

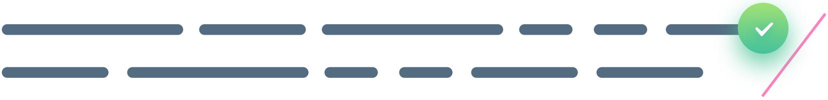


This rule is likely the easiest to implement and you can use it before you try the Golden Ratio technique. It works for most fonts well, and it's also quite good for setting the margins and paddings accordingly.

Using this rule it's best to go with a line-height of either 1.4 or 1.6 EM for all of the fonts and headings.

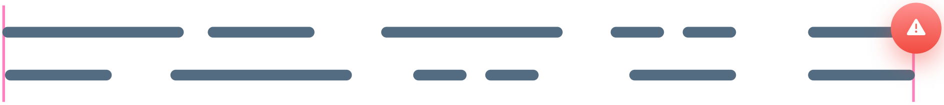
Alignment and line-length play a big role in how readable the content is (along with proper side-margins). There is a lot of rules around how to align fonts, but the most important one is:

Left-align, not-justified



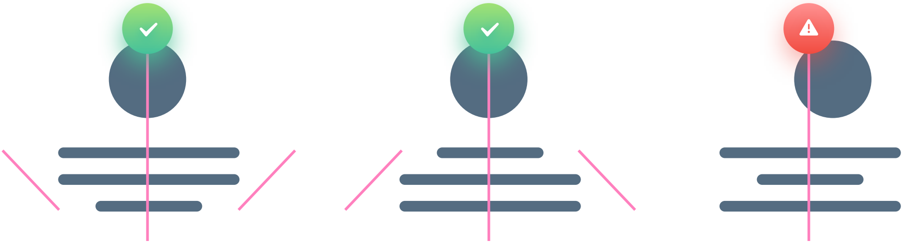
Of course it's a cultural thing and can differ between countries. Most of the western world however, uses the left-side alignment because we read from left to right. Irregularity of the right edge (called the jagged edge) of a text block shouldn't be a concern unless it's very significant. There are, however, cultures that read from right to left or from top to bottom, so make sure to accommodate that if your project will be multicultural.

Try to avoid justified text while designing digital products. Having both text edges uniform may seem friendly, but it happens at the cost of irregular kerning and varying spaces between individual words. Justified text is harder to achieve in code in a readable way.

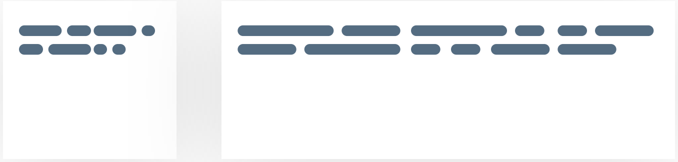


In the above example the left and right edges of text are uniform, but at the expense of completely random spacing between the words and sometimes even individual characters. This - especially for longer pieces of content is terrible for readability.

Ok, but what about centered text? It works well when it's short. You can use center-align for two or three sentence blocks. Anything longer than that will be much harder to read. Centered text also works well as a subtitle for icons or images. Try to remember to align it properly along the Y-axis of the figure. The best shape of the text is a triangle, as it makes it easier to read than an irregular shape.



The first two examples above are triangle shaped and the center of the icon goes through the center of the text. The last example is an irregular shape and the icon is too far to the right making this part a lot harder to read.



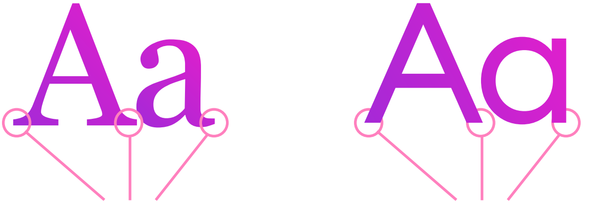
LINE LENGTH

If you're designing text blocks for mobile devices, try to keep within 30 to 50 characters per line. That will help with achieving legibility on a small screen.

On larger screens (like tablets and laptops), the best range is between 6 and 9 words per individual line. This method should help you pick the right font size for a specific text block.

> “Keep text blocks at 4-5 words per line on mobile, and 6-7 words per line on desktop.”

The two main types of fonts are serif and sans-serif. You can also find decorative, handwritten, stencil and more, but the first two are the most commonly used in digital products. Sans in the name means, that these fonts don't have "serifs" - which are those little lines at the ends of the letters.



These are Serifs on a font.

A sans-serif font doesn't have them.

Serifs are a tiny bit easier to read when it comes to longer text. That's why most books are created using these kinds of fonts. So if you're looking for the best font for a blog-post or a news article, you can start with a Serif.

Sans-serifs are great for everything else - titles, labels, form fields, buttons and other UI elements.

Avoid the other kinds of fonts - stencil, handwritten, decorative etc. They're almost always hard to read and mix horribly with other, more standard font-types. The only instance, where a handwritten font can make sense is an Author's name - it can give the name that personal touch some people really like.

But with fonts playing safe is always the best choice. The simpler the font shapes the better for readability. Obviously it doesn't mean you should use the exact same fonts everyone else uses - there are plenty of free and paid alternatives that retain the simple form, are super-readable and yet fresh enough to support a brand new UI.

Both Android and iOS have their dedicated fonts that should be your first choice for designing apps on these platforms. You don't have to use those fonts in your design, but unless the font you choose is free, you can face pretty hefty licensing fees. In general, app usage of paid fonts is the most expensive way to use them.

Try to use mobile system fonts in your first designs exclusively. The users already know those typefaces from all the other apps they are using, so your design will instantly seem familiar.

You can download and use both of these for free.

Try not to mix the fonts between the platforms, however. Using San Francisco (Apple font) for anything other than a Mac OS or iOS app can be against the licensing terms of that typeface.

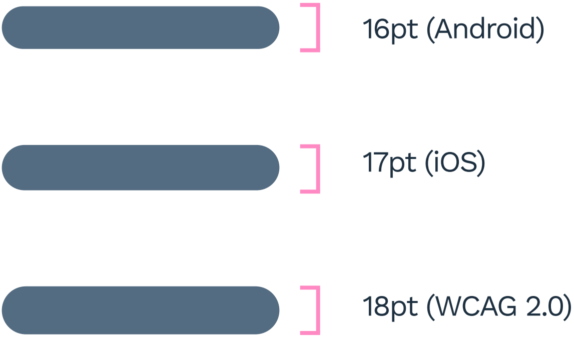
San Francisco (SF UI)

Apple (iOS, macOS, watchOS, tvOS)

Roboto

Android, Material Design (Mobile + Web)

Both Apple and Google also suggest the smallest readable font-size for body copy and inputs. Keep in mind that all fonts are different, and these values are only for the mobile system fonts like SF (for iOS) and Roboto (for Android).



It seems like 18 will be the safest, most readable option that also adheres to the WCAG 2.0 rules. Remember - people are not afraid to scroll. It's best to have larger, more readable fonts than to fit more lines of text on one screen.

Of course, it doesn't mean you shouldn't use fonts smaller than 16 in your designs. Just try not to use them for main body copy and inside your form fields. These smaller sizes work a lot better for descriptions, labels, and meta-data.

Key takeaways

- ✓ typography can make or break a project

- ✓ try to use clear rules for the size and spacing of your fonts. The Golden Ratio is a great way to start.

- ✓ never use text-align: justify;

- ✓ 1.2 to 1.4 em is the safe line-height for most fonts

- ✓ too little whitespace can negatively affect the readability of your fonts. Too much whitespace will also cause a similar effect.

- ✓ start with a base font-size and calculate all the other sizes from it.

Forms

THE BASICS

A form is one of the most common ways a user can interact with a digital product. It helps with making a purchase, creating a profile, or signing up for a newsletter. Done right can boost a business. Done wrong can break it.

In simple terms, a form is a set of labels and fields in which we manually enter information, with a button at the end, to save everything in a database. But one thing that's most important in any form is conversion. The entire focus of building it should go towards conversion, not the technologies used or the design. Of course they add to the conversion, but they are only means to the end.

So what is conversion? **Conversion** is a percentage of completed form entries. Every user that "drops off" before clicking that final "save" button is a decrease in the conversion rate.

It's the job of the UX designers, UX writers and stakeholders to figure out which fields are unnecessary, how to clearly communicate what we want from the user and how to guide their focus through the form.

It's a mix of these factors that contributes to a great form-filling experience and on top of that it has to be coded in a way that adds even more engagement to it. The unnecessary fields should be removed because if something is not needed, then making the form longer will only increase frustration - people will get bored and drop off. Clear UX copy is necessary because we want them to understand what information we need. And great design and readability are essential for guiding them quickly through the task.

But don't think developers can't influence the conversion and the job of making the forms better should be in the hands of the designers alone.

There are many things you can do to make it better, and these are essential to show your worth as the developer. Understanding this will lead you to progress in your organization quicker, or make your startup a lot better.

The areas I believe you should focus on are field validation, alternative input methods and chatbots. Let's briefly go through why they matter.

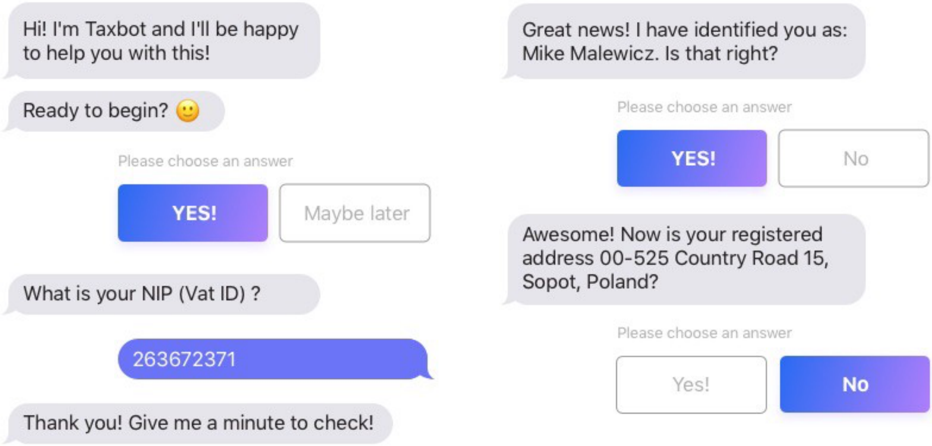
Field validation seems simple, right? You simply check if the email address they entered has a @ in it and call it a day? No. The possibilities go way beyond that, including automatically fixing fields like zip-codes, phone numbers etc. Don't expect your users to format them just right, be ready for them to write either 01-903, 01903, 01 903 and other possible combinations.

As long as the numbers are right, it should be your job to write it to the database in a way the system will understand. And while we're at it, if they entered a zip-code, you can pre-fill the city, county and more based just on that (with an option to edit). Think of it as a challenge to see how little user data you really need to fill out the form. Then figure out the most elegant ways to achieve that.

Alternative input methods are a bit more tricky, but giving the users a choice is really important. The form should easily be accessibly with touch, mouse-input and using the keyboard only. You can use arrows to switch between fields, bringing the current one into focus as an example of a good keyboard-based input. The more the user is focused on one task - writing - the easier it's for them to fill in the right information. Making them move their wrist to a mouse or a trackpad after every field is adding friction.

> “Developers can also contribute to better form conversion with Field validation, alternative input methods, chatbots and more! ”

Chatbots are the future of longer, more complex forms. Contrary to the popular belief, they don't need to be AS smart and AI-powered as most developers think. Sure, it's nice if they can understand more, but in the case of many forms it's simply asking the right questions and providing the right ways to answer. Look at the example below.


Hi! I'm Taxbot and I'll be happy to help you with this!

Ready to begin? 🙂

What is your NIP (Vat ID) ?

263672371

Thank you! Give me a minute to check!

Great news! I have identified you as: Mike Malewicz. Is that right?

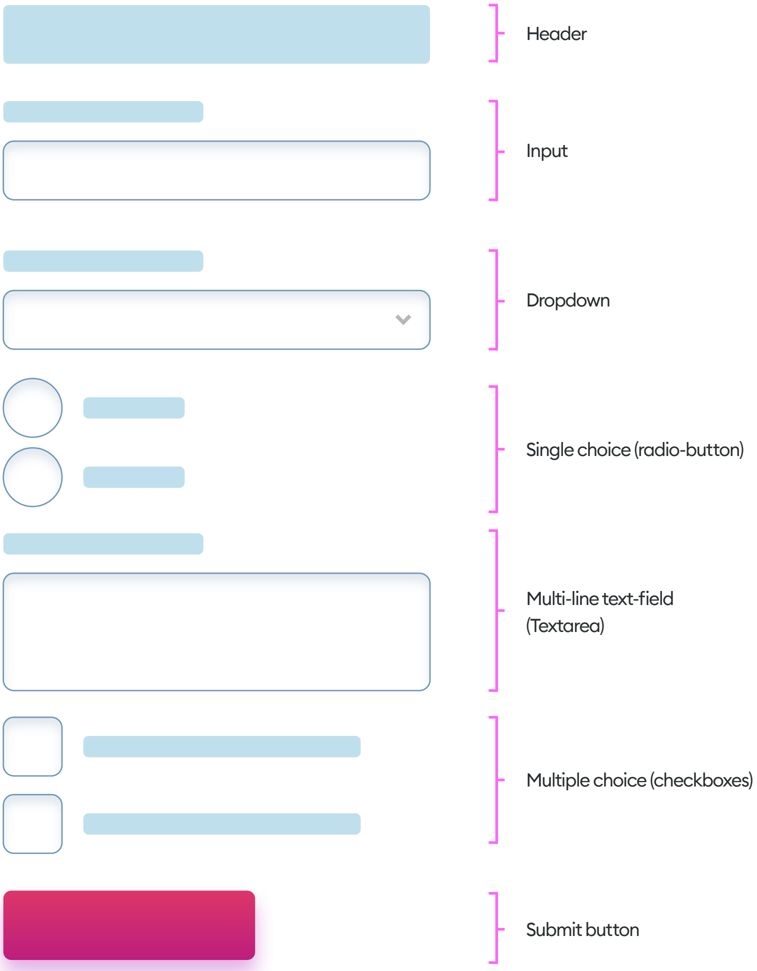
Awesome! Now is your registered address 00-525 Country Road 15, Sopot, Poland?

This is a chatbot prototype we've created to handle tax return forms. Wherever we can we limit the user input to just buttons, and only request them to write anything if it's absolutely necessary. If you have more than two choices, you can present them with a dropdown too. That way you have a nice, conversational interface that doesn't require any AI. Most of these interactions can be done on the front-end side alone.

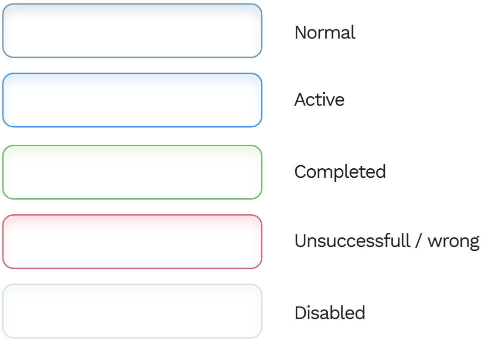
Understanding chatbot mechanics is one of the ways to stand out from the crowd of front-end developers. It's essential to understand, that the technology is only a path here. The path leads to a much easier, faster and more fun experience of doing something otherwise boring and horrible.

FORM ELEMENTS

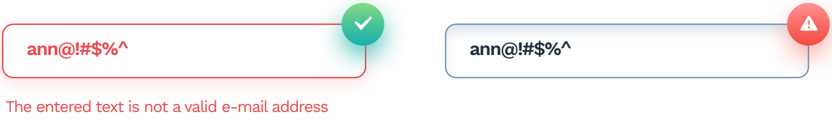
The basic set of elements in a form includes a **text field**, a **dropdown**, and a **button**. We also have multiple-choice fields (**checkboxes**) and single choice ones (**radio-buttons**), text-areas for longer text entry. Other types (like sliders) are less common. When coding your first form try to cover every one of these essential elements first. These are the ones that you need to master!


Header
Input
Dropdown
Single choice (radio-button)
Multi-line text-field (Textarea)
Multiple choice (checkboxes)
Submit button

A form field can have one of five possible states. When coding, try to differentiate the states for every form element (field, checkbox, radio, etc.) The easiest way is starting with the generic text-field (input) and then move the shadow, border, font, and color styles over to the other elements.



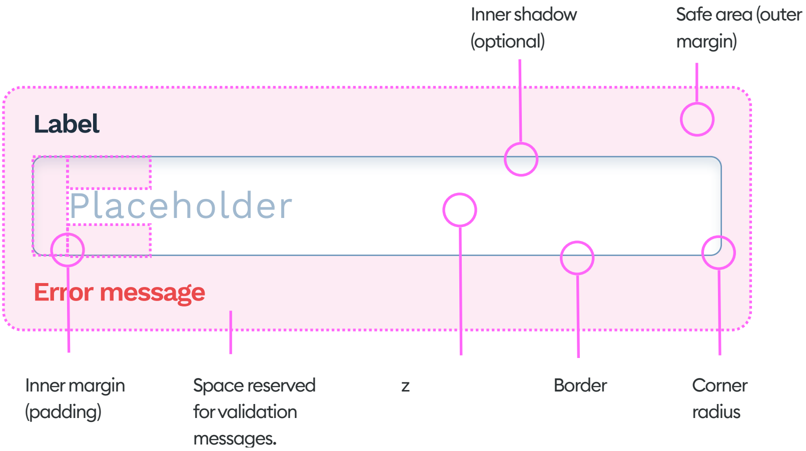
The differentiation is very important, but avoid using only a color to do so. It's best to also have a little text under the field for better visibility.



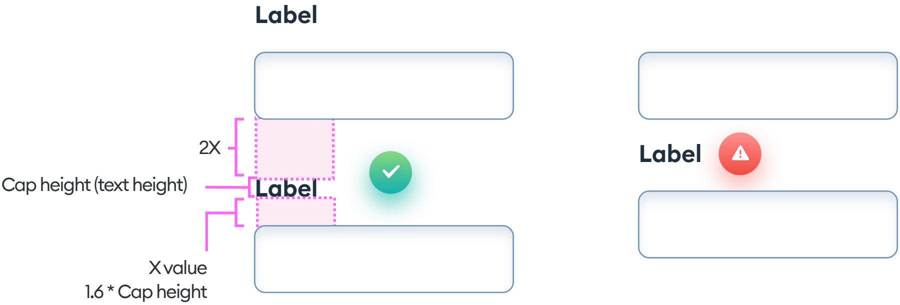
The entered text is not a valid e-mail address

Avoid having all validations displayed under the entire form, or inside popups. Keep your descriptions brief and to the point. It also makes sense to show the proper way to fill the field as an example. If your designers didn't prepare those extra form states, this is your time to shine! Wow everyone with thoughtful and easy to explain tips on better forms!

The most common form element you'll likely be working with is a text-field. It has a couple of characteristics that you need to be able to replicate well in code even if your designer took some of them for granted. A good field needs most of these to perform best.

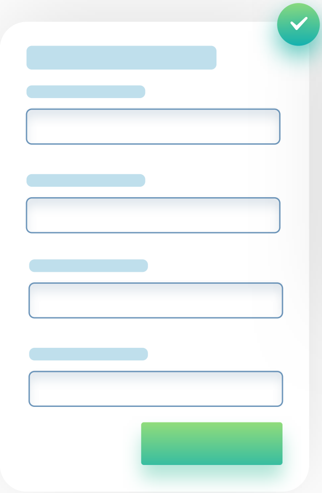

Inner shadow (optional)
Safe area (outer margin)
Label
Placeholder
Error message
Inner margin (padding)
Space reserved for validation messages.
z
Border
Corner radius

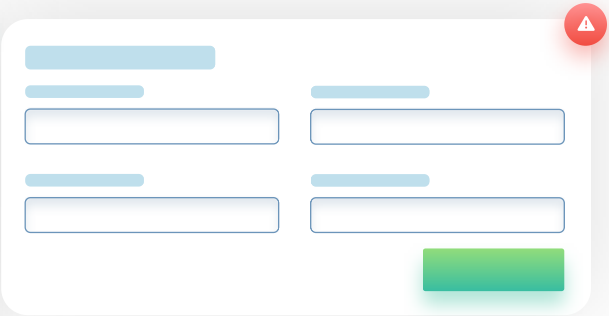
Label alignment is also an essential part of a great form. You need to be sure that the label is ALWAYS visibly closer to the field that it belongs to. Labels placed perfectly between the fields are one of the most confusing things about any form.


Label
2X
Cap height (text height)
Label
X value
1.6 * Cap height
Label

SINGLE VS MULTI COLUMN FORMS

Single column forms convert a lot better than multi-column ones because you only have a single path to follow with your eyes, instead of jumping between blocks.





The example on the left will work a lot better because it only has a top-to-bottom scan line to follow.

Keep forms rather narrow. The best range is somewhere around 320 and 500p wide (mobile and desktop). The wider the fields, the longer it takes to process them - keep it simple!

Longer (6 fields and up) forms are better divided into separate pages, with clear information on which step you currently are.

Don't mind the empty spaces on the sides of your form. Don't fill it with placeholder images or decoration. Leave it empty! It's great for giving the form maximum attention and the best focus.

“

Single column forms convert better than two-column form layouts.”

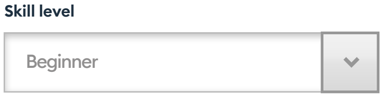
THE SELECT PROBLEM

Selects, or "dropdowns" are one of the most tricky form elements to code. Sure - they're rather easy to do in their default state, but there are a couple of problems with that approach. The first one is that many operating systems, or even browsers render the end result completely differently. And while it makes a little sense - people are used to select boxes from the OS they use, it makes the product less recognizable when they switch to another device.

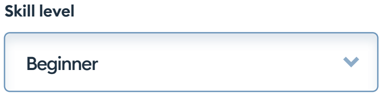
The second, more trivial problem is that most of the default styling options are simply ugly with not enough whitespace, poor choice of selection indicators and less than optimal scrollbars.

Most developers I know create their own set of styles for that component and only modify the defining characteristics from project to project. These include changing the corner-radius, outline-color, font or the type of up/down arrow at the end of the box.

It's important, however, to always remember to keep that element native on mobile devices. While it's good to have that consistency and beauty on desktops, mobile "spinner" paradigms for select boxes work way better on a smaller, touch enabled display.



Standard, generic system dropdown.



Custom dropdown, requires additional coding.

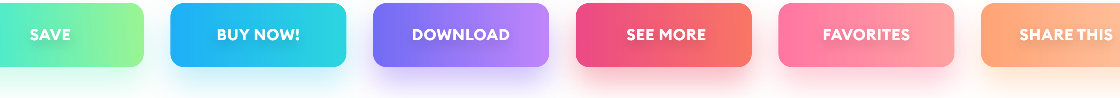
Key takeaways

- ✓ Conversion is the key metric when it comes to forms.
- ✓ Make sure it's clear which form field a label belongs to with proper spacing.
- ✓ Developers can help with form conversion with great validation, inteligent pre-fill suggestions, alternative input methods or chatbots. Use this to your advantage.
- ✓ Show validation errors right next to the form field they relate to - not below the entire form.
- ✓ Narrow, single column forms with not too many questions convert best.

Buttons

THE BASICS

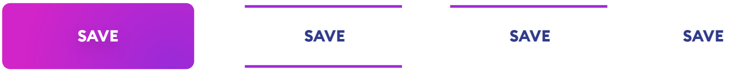
A button is an interactive element that results in an action described in it. You can bet that if it says "save" on a button, clicking it will most likely "save" something. When it says pay, it's likely that the next screen will be a credit card details page. It's also an essential interactive element of a digital product. It's crucial to get buttons right.



It can lead to a purchase, downloading, sending, and many other vital actions. Digital buttons are also descendants from real-world buttons like in the tv remote, record player, or game controller. In this chapter we'll cover the very basics of how buttons work and what you should look at carefully while building them in code.

BUTTONS SHOULD LOOK LIKE BUTTONS

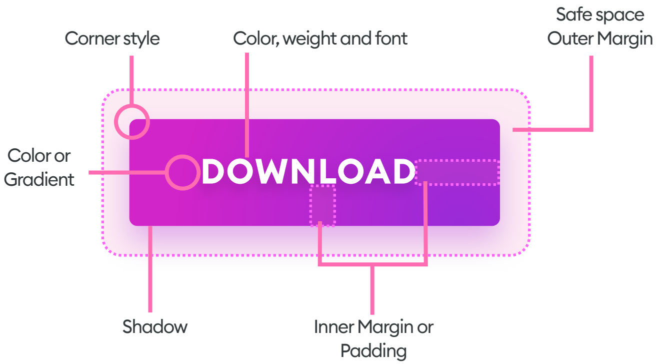
he most important rule while designing a button is for it to stand out enough so it won't be confused with anything else.



By removing elements from a button, it's function starts to dissolve and disappear. It becomes decoration or text, losing its actionable qualities.

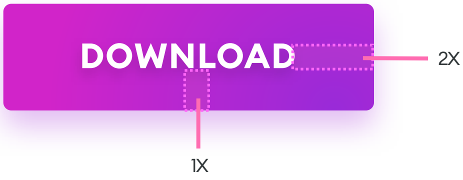
> “A button
> should look like
> a button.”

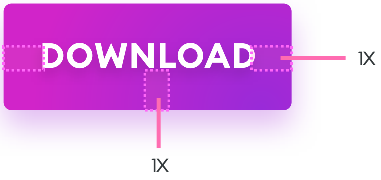
While building buttons, remember about all of these elements and check, whether the design contains definitions for each one. When in doubt you can ask the designer. If you don't have a designer onboard, you can follow the guidelines from this chapter, to be able to make the most out of your buttons.


Corner style
Color, weight and font
Safe space
Outer Margin
Color or
Gradient
DOWNLOAD
Shadow
Inner Margin or
Padding

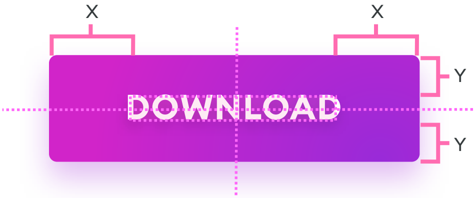
You should set the padding and the safe space using your grid base numbers. Make sure your button has enough margin (or safe area) so it's never too close to other objects. The corner style should be consistent with that of form-fields or other on-screen elements.

In the example below, the left inner margin is twice the size of the vertical one, which is a safe choice for increased readability. When in doubt you can use that rule, or the 2W rule I outline on the next pages. Keep in mind, that the smallest possible value on the sides should be 1X and not smaller than that. The button on the right is thus the smallest padding you should be using.

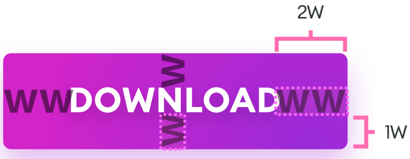

DOWNLOAD
2X
1X

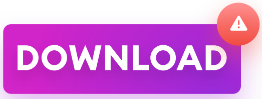

DOWNLOAD
1X
1X

Poorly aligned and unevenly spaced buttons are one of the most common problems of all interfaces. Double-check if your button labels are centered both horizontally and vertically. You can take screenshots and then measure the number of pixels below and above the Cap-height of the label.

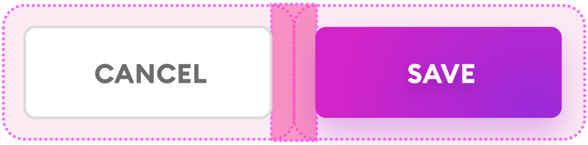


Aside from grid-based methods, there's a safe way of choosing button spacing with multiples of capital W. If **at least 1 W** fits on each side of the button label, you're safe. On the sides, it's even better to use **2 x W** for increased readability. In general, the smaller the padding inside the button, the harder it is to read and understand. Avoid buttons looking like the example on the right!



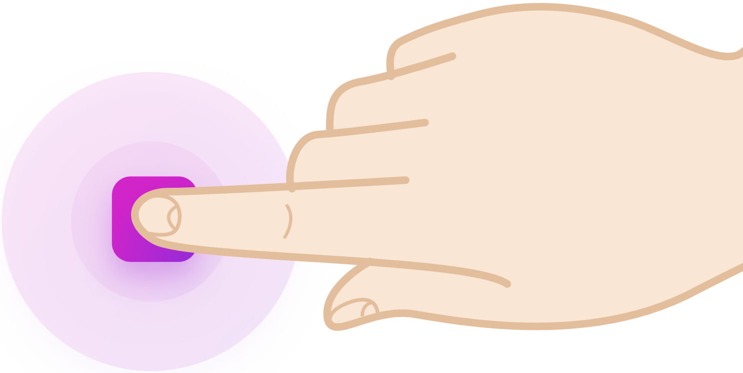


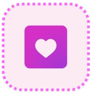
Don't forget about the space around your buttons. If you have a group of them, the safe area (or margin) should be individual for each one - try not to overlap it!



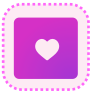
Both web and mobile buttons should also have a set minimum size. If your build your buttons to be too small, it will be difficult to tap or click on them. That results in frustration and can lead to users uninstalling your app. The best way is to start with 44 by 44 points for all interactive elements on mobile devices. The sweet spot is somewhere around the 50p height for mobile buttons. The width (including the Double W minimum spacing) can be any number equal to or larger than the height.

In the case of cursor-based devices, 32 by 32 should also work. Remember that even on desktops, the larger the button, the easier it is to use.

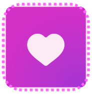




This button is too small. The inability to quickly tap or click on it will frustrate your users.



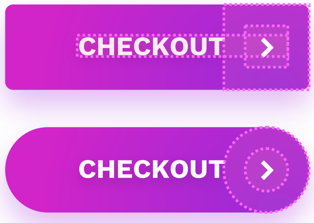
This button is still a bit too small for mobile devices. It probably will be fine on laptops.



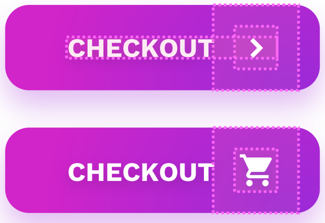
This button is big enough for both computers and mobile devices. People should be able to find and activate it with ease.

To make your button stronger, you can add a right facing chevron next to it. Make sure, however to align it right.

Aligning icons on buttons is one of the hardest tasks for both designers and developers alike. In many cases, it depends on the font-weight to icon weight ratio and their "optical size." However, you can try and use this simple solution that will work fine in most cases. When building these buttons try to create the containers with the actual backgrounds first - for better alignment and then remove the backgrounds when it's done.



Create a square or a circle the size of our button height. If it's a pill or rather sharp-cornered button, just align it to the right side of the shape. Then create another smaller object inside of it with its spacing being the height of our text. Then fit the icon right in the middle and try not to make it larger than the smaller circle. In the case of chevron, text-height works for chevron height as well.



If our border-radius is larger than 8 points (but not quite a pill), it's best to align our square to the place where the border starts to bend. If you need to use a more complex or larger icon (like a shopping cart), make sure it's not bigger than our smaller container.

As a front-end developer, you'll likely be making A LOT of buttons. It's simple - they're the most common element that causes action. You can't purchase anything online without first pushing a button or two.

I believe there is a tremendous space to grow here, as most developers still didn't figure out buttons that well. It all starts with the shift of the mindset. A button is not "just" a rectangle with some text inside. Think of it as an essential element of the interface. Sure - rectangular shape and a well aligned label are enough for a button to work. But what can you do above the standard to stand out? Think of the button as an entire project inside a larger one.

Making nice, animated state transitions is one thing that instantly comes to mind. Cool ideas include flipping the gradient on hover, or even slightly darkening each color, but with a timed transition. This will make it look more polished, fluid and generally nicer to use.

If you plan on adding a chevron, you can also animate it slightly on hover with the arrow slowly moving left and right by a couple of pixels.

You can experiment with making the button smaller or larger (or even bouncy) on click or hover too. Just remember that enlarging the text as an animation can often be problematic on some devices / browsers. But you can go around this and just enlarge the button shape, leaving the text intact.

Just remember - animations have to be subtle (small movements) and not necessary to understand how the button works. They can supplement or reinforce the message, but should never be used as the message itself.

Key takeaways

- ✓ Buttons have to look like buttons!
- ✓ Avoid "crazy" button shapes like blobs or triangles. The best button shape is a rectangle, then a text-link and then a circle.
- ✓ Make sure your labels are both horizontally and vertically centered on the button shape. This is crucial, as almost nobody does it right!
- ✓ Labels on buttons need to "breathe" - make sure there's at least space for one capital W on each side of the label.
- ✓ Best minimum size of a button is 44x44 points on mobile and 32x32 points on desktop.
- ✓ Bigger buttons are always easier to click.

5.

You're the designer

You're the Designer

DEVELOPER LED

It's possible that your company doesn't have a skilled UI designer among its ranks. Or in a small, technology-driven startup you simply can't afford one just yet. Don't worry!

Besides, front-end development is very close to UI design anyway. Just think of this whole design thing as creating containers of the right size and in the right place.

WHEN THERE'S NO DESIGNER, IT'S UP TO YOU TO BECOME ONE

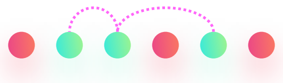
In the previous chapter I walked you through the most essential parts of design, so now it's time to take some of that knowledge and put it into practice.

In this chapter I'll cover a couple of basic exercises from how to best set the margins and paddings, all the way to pairing fonts and fixing ugly layouts. Keep in mind, that these techniques are "safe", but they don't solve all of the potential problems. But if you plan to build a product alone, without a designer, try using these rules for the best result.

You can use this chapter as a guide and come back to it every time you're playing the role of both designer and developer.

But don't worry - you can "design" in code so no need to learn any design tools. It's all about having a plan and knowing what you want to accomplish.

The main rule in implementing a good design is to have all the objects visually "connect" with each other. That sounds super philosophical, right? No worries! It means that all fonts you use should be proportional, colors should match each other, while margins and paddings should define a clear, easy to understand hierarchy. It's like connecting the dots - everything lies in knowing which ones to avoid.



The hard part is to figure what you won't need and what you definitely will.

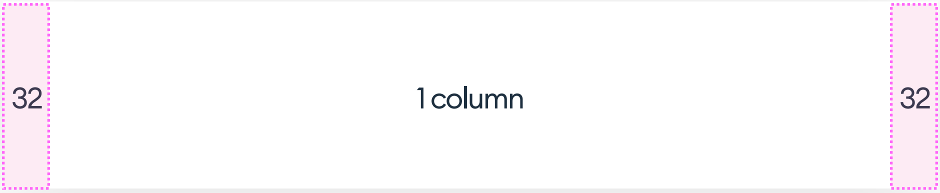
That's mostly because you can't predict every possible interface type beforehand. There will be times when you'll be tempted to add a font-size: 11px in between your already defined 10 and 12.

Avoid the temptation and be strong!

Of course consistency for the sake of consistency isn't the way to go, but good interfaces are consistent for a reason. It's because your users looking at a coded UI can "expect" certain elements to look a certain way. That frees up some of their cognitive power and they fly through your forms and pages much faster and more efficiently. That efficiency and ease of use is what you should be looking for.

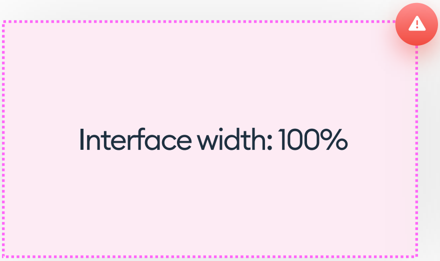
> “Consistency and small scale of your UI component set is crucial for easier understanding of it by the users.”

Unless you're working on a very dense, data-heavy interface, you can use the concept outlined below to create a grid that will fit your product.

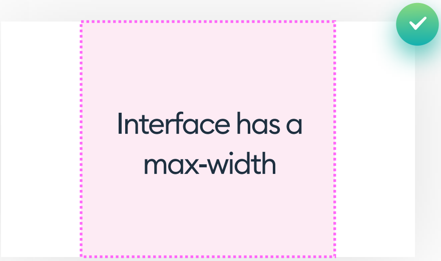


Start with defining your side margins again. Let's go with 32 points. If you're working on a mobile project, knowing that the margins are 32 is enough. If it's for desktop you need to figure out how wide the interface will be.

It's essential to set this to 1300 pixels at the most - even if you have a much larger display. Most projects will not benefit from all that extra space, and it will make it harder to read because we'll be breaking the "words per line" rule outlined in the typography chapter. With over 1300 pixels there is no way to achieve 6-7 words per line unless they're a grotesquely large font.



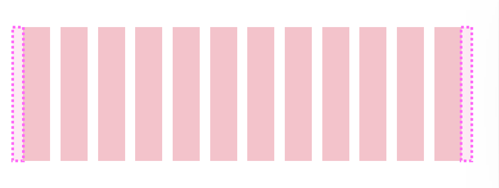
2560px wide display



2560px wide display

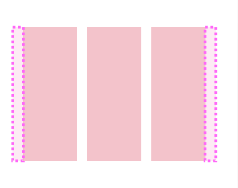
On larger, 27" displays you often get resolutions larger than 1920p - so obviously stretching the interface to the full 2560 or more is simply a bad idea. It will be too wide to be readable.

The next step is to determine how many columns will your interface have. You can go versatile with the example grid outlined in the previous chapter, but if you only plan to have one column, it may be better to actually create a grid dedicated to it.



Desktop

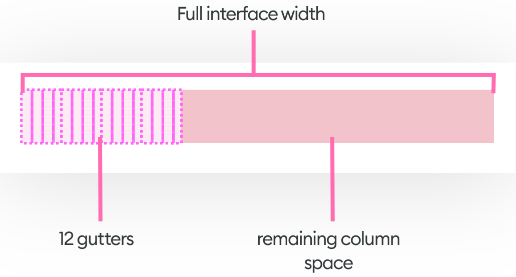
Width: **1220**
No. of columns: **12**
Column width: **80**
Gutter width: **24**



Mobile

Width: **100%**
No. of columns: **3**
Column width: **Auto**
Gutter width: **24**

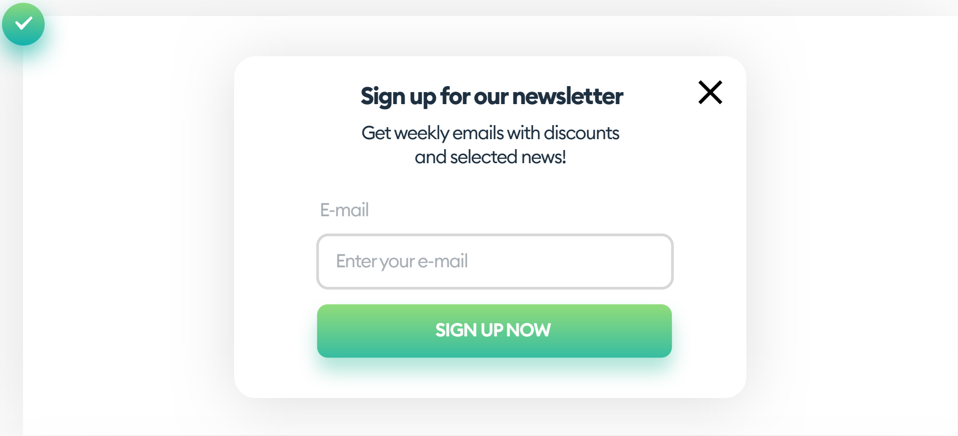
You can use an online grid generator, but at first it's best to try and actually calculate it yourself. Just select a gutter value (24 or 32) and then multiply it by the number of columns you plan to have. In case of 12 columns it's simply 288 px and 384 respectively. Now find a number between 1200 and 1300 that divides evenly by 12 after you substract the gutter value.



This is how grids are calculated, so it's best to understand it before playing with online generators.

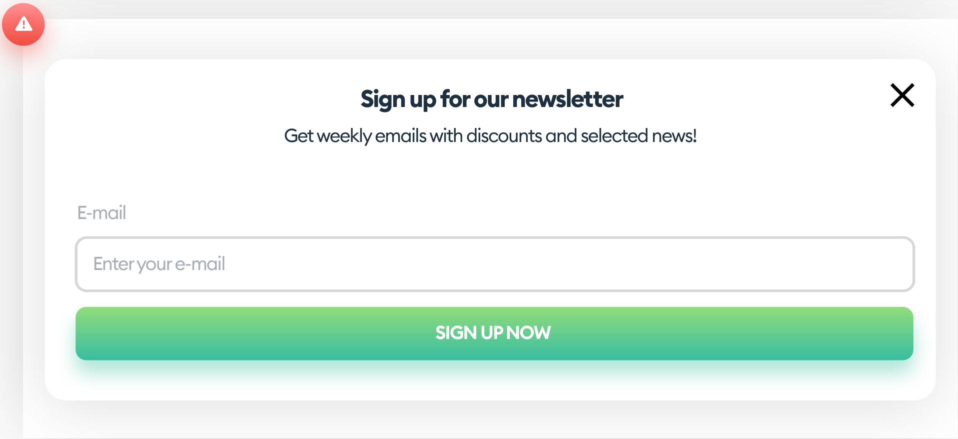
PRE-MADE GRIDS

There are many popular grid systems out there - with **Bootstrap** being currently the most widely used on the web. The problem with developers using them is that they often don't really think about how the grid works, and what number of column is right for any given interface. That leads to some windows being way too wide, or too narrow because a "standardized" grid was enforced without thinking. The example shown below is sadly a lot more common than we'd hope for it to be. And it all comes down to the front-end developer not picking the right number of columns for the screen.



Sign up for our newsletter

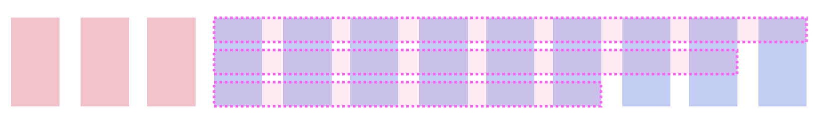
Get weekly emails with discounts and selected news!



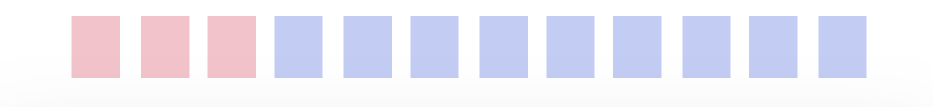
Sign up for our newsletter

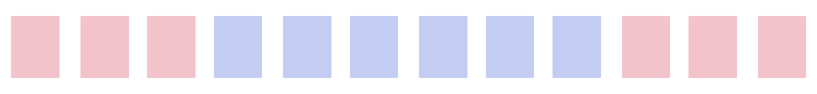
Get weekly emails with discounts and selected news!

If you're building for the World WIDE Web ;-) you need to understand how wide or how narrow certain UI elements can be. We already discussed, that a good text-block usually has about 50-70 characters per line. Knowing that, and the fact that 14p font-size is the smallest you should use for blocks of text, you can easily calculate how wide your line of text can be to still be comfortable. It usually means it won't be as wide as the entire layout. That is unless you plan on using a very large font size but that has its downsides too. When working with a 12 column layout you can use a range of 6 to 9 columns for longer text.

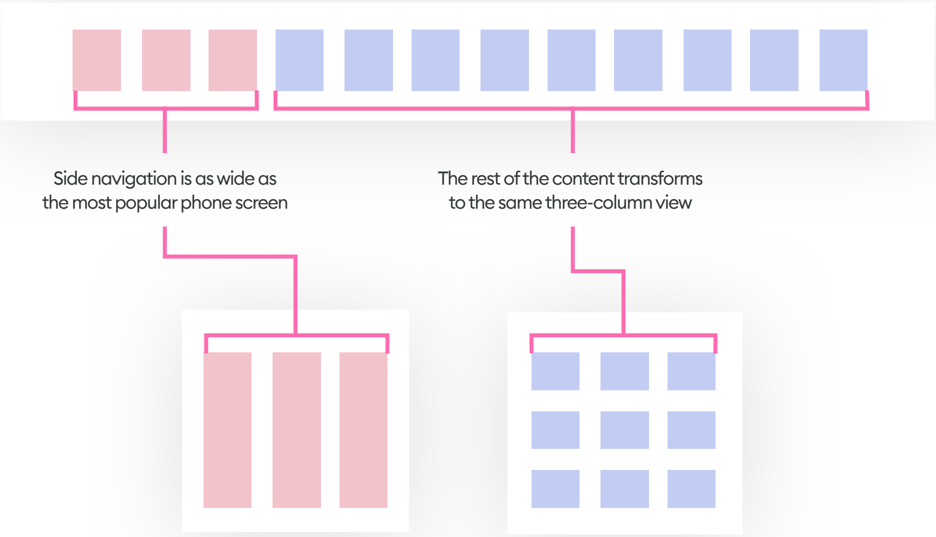


Some elements however, like a filter bar or a side-navigation menu shouldn't exceed 3-4 columns. Making them wider will make the content within them more difficult to scan and thus harder to use. This gives you a couple of potential layout options showin in the examples below. The blue columns show the "body" of the page - like a blog post, article or any other form of content. The red columns show navigation and sorting.





Another good idea for making responsive layouts is making the desktop navigation panels the width of a typical phone screen. In most cases they work as a two-step process anyway (while on desktop you see them both at the same time). Doing it that way means that the side-columns won't have to change a lot when displayed on mobile devices.


Side navigation is as wide as the most popular phone screen
The rest of the content transforms to the same three-column view

If you're working with a two-sidebar layout you should of course keep the other one three columns as well. The main difference is that on a mobile device you only see the navigation screen (red columns), and tapping on a link from it you then go to the content screen (blue columns).

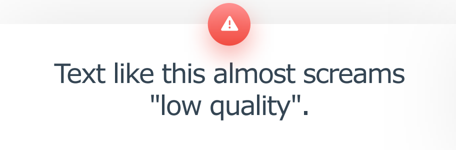
Remember to include a back arrow on the top-left of the content layout and this pattern will feel very intuitive to your users. Another important change between the desktop and mobile versions is font-size and touch target on links and buttons. On mobile it should be in the range of 44-48 points or more, while on desktops it can be lower at around 32 points.

> “Each UI element has a minimum and maximum comfortable width. Make sure to learn the right patterns.”

Unless you're making an art-installation website, you're going to use text to deliver most of the message of the product. What we've noticed is that many developers pick the most common, obvious fonts for their projects.

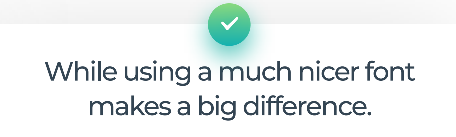
While going the safe route is not a bad idea, picking a slightly less popular font will make your work stand out a lot more. Fonts really do make a difference and you should leverage that to stand out.

Let's start with what NOT to use.
Fonts like Arial, Verdana, Tahoma (and generally all Windows 95 era fonts) are a bad choice for modern digital products. They were made back when most displays barely reached the 72ppi screen densities and can lower the perceived value of the project.



Text like this almost screams "low quality".

Verdana



While using a much nicer font makes a big difference.

Montserrat - Free font

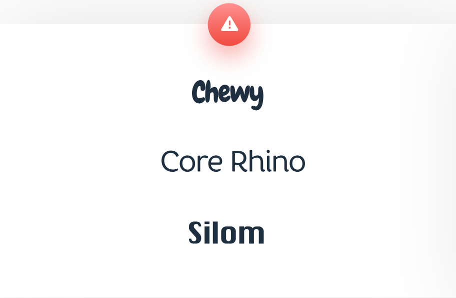
The main thing to consider when choosing a font is whether you, or your organization is willing to pay for it. There are of course many great, free to use fonts out there too. But finding a paid one that's also good will likely mean it's less commonly used. It's a great, and often cheap way to stand out. There are both monthly subscriptions (like Typekit) and fonts you can buy and own forever - regardless of the number of views your product gets.

A free, but less common custom font will also do the job, but the better the font, the more popular it slowly gets, so you may end up changing it after a while to stand out from the crowd again.

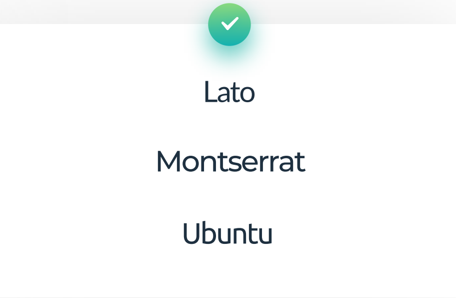
The "beauty" of a font is something only a handful of designers really understand and appreciate. Obviously its looks are completely subjective, but don't pick a font that only you will like - it has to work well for a wide variety of users.

There is one rule that you should always try to follow for the base-font on your projects. It has to be simple. No varied line-weights, no exaggerated roundness or funky skew. Helvetica became so popular because it lay the ground rules for how a readable font can look like. Of course every font has its unique twist, but you need to make sure the twist isn't too "in your face".

Here are some examples of sans-serif fonts:



These fonts are a little bit "too original" which makes them harder to read and process.



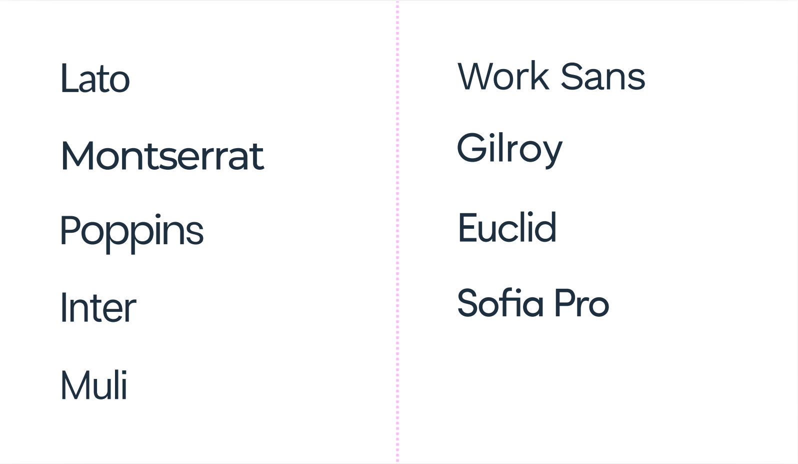
Here are three good choices, that are also completely free to use and a lot easier to process.

Look for that simplicity in fonts. Notice how the group on the right is visually coherent while still having their own "perks" like the letter "b" in Ubuntu. This is the level of "funky" that you should aim for.

An experienced designer could of course (in certain product types) use fonts like Chewy or Core Rhino and still make it work - but we're talking safe choices for a developer here - the other group will yield you a much better result.

SOME GOOD CHOICES

There are literally hundreds of free fonts you can choose from. If you don't want to go the standard **Roboto-route** most developers take nowadays, here's a list of fonts that we recommend you try:



Lato
Montserrat
Poppins
Inter
Muli

Work Sans
Gilroy
Euclid
Sofia Pro

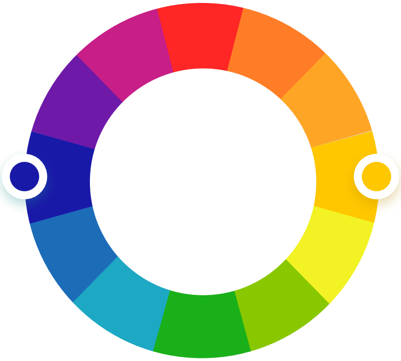
Look for that simplicity in fonts. Notice how the group on the right is visually coherent while still having their own "perks" like the letter "b" in Ubuntu. This is the level of "funky" that you should aim for.

An experienced designer could of course (in certain product types) use fonts like Chewy or Core Rhino and still make it work - but we're talking safe choices for a developer here - the other group will yield you a much better result.

As for the free Serif fonts (great for body copy on blog posts and articles) you can use **Charter, Times, Times New Roman** or **Georgia**, as they hold up surprisingly well. Consider pairing one of them with a sans-serif for titles and UI elements.

“Don't use the basic
fonts everyone
else uses.”

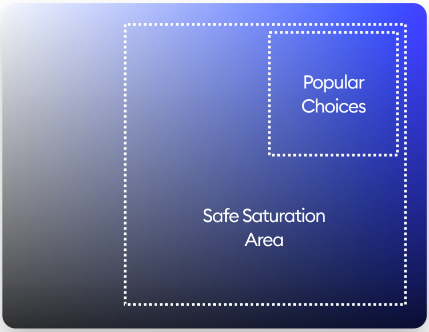
Colors are simple - until they're not. In the previous chapter we talked about the different types of palettes, so it's about time we put that knowledge into practice! If you don't know what you're doing yet here's a quick, safe example of creating a color-scheme that works. We're going to use a complementary palette, but you can use the same concepts and pick another one like triadic or split-complementary.



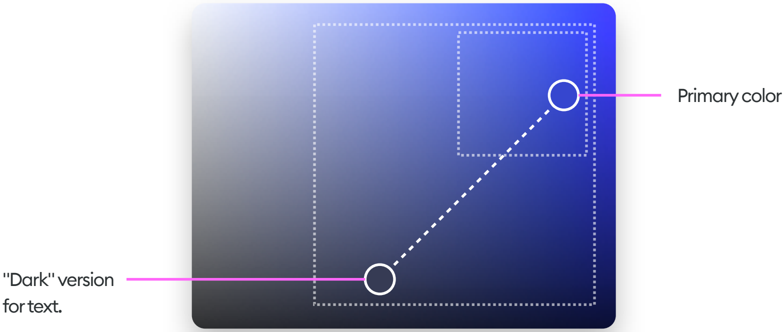
A complementary color palette simply uses two colors on the opposite sides of the color wheel. It makes it a lot easier to pick the second color based on the one we picked as the primary.

But before we proceed with picking the primary color let's talk about color saturation first. Skilled designers can work with super-saturated colors (at 90+ Saturation) by limiting their impact with other colors, fonts or on-screen decorations. It is, however, very tricky to use these super-saturated hues and many designers and developers who try it fail miserably. The problem is of course with balance - if the colors are too strong, they'll simply put the entire design off balance and make our eyes strain. Let's not do that. We want a pleasant, easy to look at experience from the start.

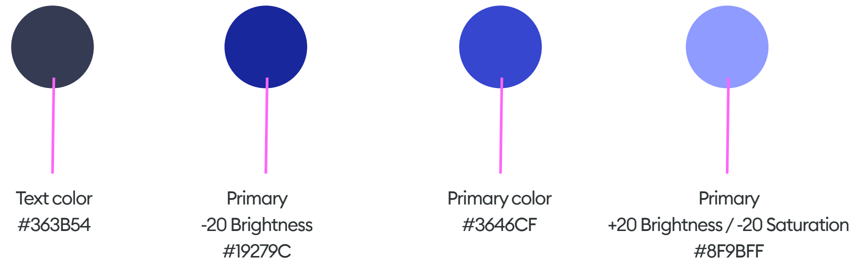
Using our handy saturation limiter below, we can pick ourselves a blue (or another color if that's your brand). The Safe Saturation area shows where the colors will be easy to work with. Popular choices show that when most people pick a blue of that hue they usually pick from that range.


Popular Choices
Safe Saturation Area

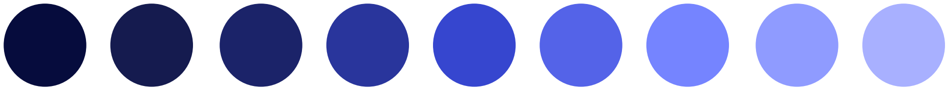
As we want our project to have a bit more "energy", I'm going to pick a spot among the Popular Choices. Then using a 45° angle I draw a line to pick a complementary dark-blue-gray that is going to work great for all the text colors in the project. The popular choices area is popular for a reason - these saturation/brightness levels make the colors pop.


Primary color
"Dark" version for text.

With this quick trick we have two matching colors that are a great start of a project. But to make this more versatile we also need a lighter and darker hue of the primary color. It will come in handy with hover effects, inactive states and more.

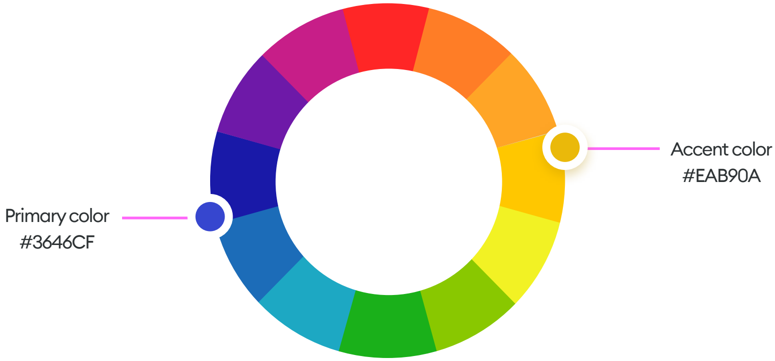

Text color
#363B54
Primary
-20 Brightness
#19279C
Primary color
#3646CF
Primary
+20 Brightness / -20 Saturation
#8F9BFF

You've probably seen the Material Design color palettes, or ones done for CSS frameworks before. They have dozens of shades of each color. Do you need to have that many as well? Not really - in most cases those systems show you a lot of options because they can't predict what you'll be using it for. But using the method of adding or reducing brightness and saturation you can easily create additional "levels" if you need them. Initially it's best to start with just four primary shades and white. We already mentioned not to use pure-black (#000000).

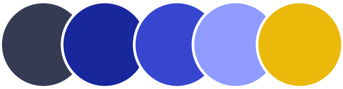


It's highly unlikely that you are going to need that many shades of primary color, unless the project you're working on is extremely complex.

The complementary hue is best for accents. It means that most of your interface should be done with greys and shades of your primary color, with the most important elements being highlighted using the accent. You can use it for a Call to Action button, active tab selection and more. Keep in mind, to use that second color only for these important tasks. Don't add it everywhere!



While picking the color on the other side of the color wheel the only important aspect you should look for is similar levels of saturation and brightness. If they match then the colors will most likely look great together. For accents you don't need to create those extra darker and lighter hues however. So just like that are very first color palette is created. Congrats! Use the guides from the previous chapter to create something truly unique and then stick to that with meticulous consistency!



A simple, complementary color palette that could be a great starting point for your project.

> “You don't need every possible shade of your primary color for your project. Pick a few that work best!”

6.

Nudge the designer

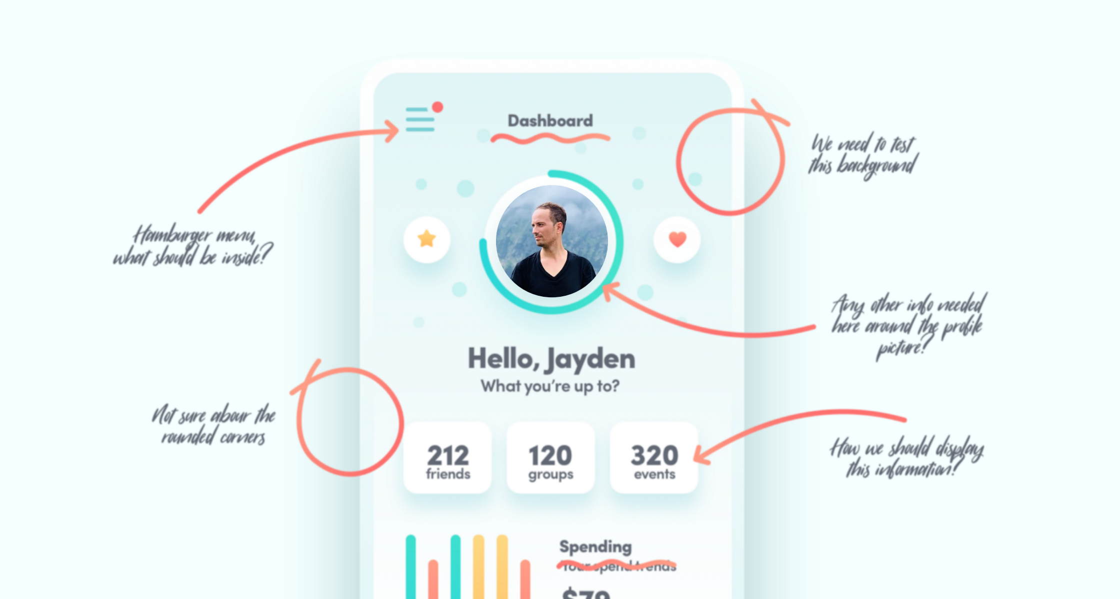
Nudge the Designer

DEVELOPERS AND UX

What I'm going to say now may be a bit bold for some, but that's something we learned over time while working on multiple projects, be it mobile apps or websites. Developers are in part responsible for the User Experience of the product, just like thedesigners are. Certainly not to the same extent, but still.

As a developer, you are not going to design how the interface should look like, nor come up with the text copy, but you have to know about the impact your work has on users' perceived experience interacting with the product.

We've seen too many developers not paying enough attention to design, to avoild talking about that. That's one of the reasons we decided to write this book. It leads to thousands of apps with a misaligned interface just because developers were sloppy and didn't bother to make this one more step to make it look or work better.


Dashboard
We need to test this background
Hamburger menu, what should be inside?
Any other info needed here around the profile picture?
Hello, Jayden
What you're up to?
Not sure abour the rounded corners
212
friends
120
groups
320
events
How we should display this information?
Spending
~~Your spend trends~~

Good developers should know how to work with designers so that they can make better products together. You can help them with your knowledge of common patterns that you encounter, and they can make your life easier by providing assistance and explaining what's behind their decisions. Our experience tells us that mutual understanding helps streamline the whole process by a lot. You can get your app delivered in less time with improved quality.

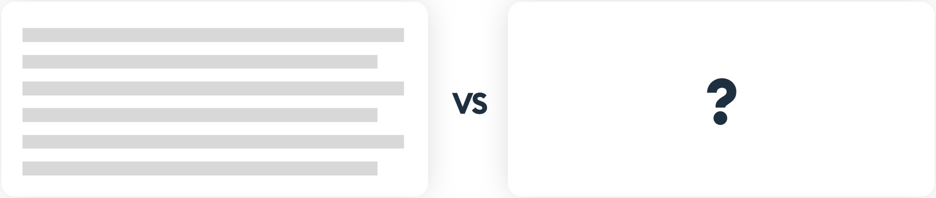
Getting lost in the creative vision is a trap designers happen to often fall into. They get focused on the big picture and visual side so much that they can forget about other details, which can be crucial for coding a real product based on their design. You can't really blame them; it's not a trivial task to think of every possible interaction and come up with a detailed answer for how it should look like. But you shouldn't be passive about it and decide you are going to use a generic solution to that problem because the designer did not cover it. You'll make a much better impression on your teammates if you're the developer that can talk to designers and make it meaningful for both sides.

That's why sometimes you have to step in and let them know that they didn't think about something and you need their help. User Interface is there to make users feel good about using the product, and you are responsible for making sure that it works as expected.

We want to present a few common scenarios that remain forgotten by many designers but can help build a great user experience when appropriately designed, without leaving it to developers will. You can use this as a checklist before starting coding to see if designers provided everything you need.

UNUSUAL STATE

Under this category fall all the states that your app can be in, due to certain circumstances, like slow internet connection, an error returned from the server, or a fresh account without much information to display.



AN EMPTY LIST OR ZERO STATE

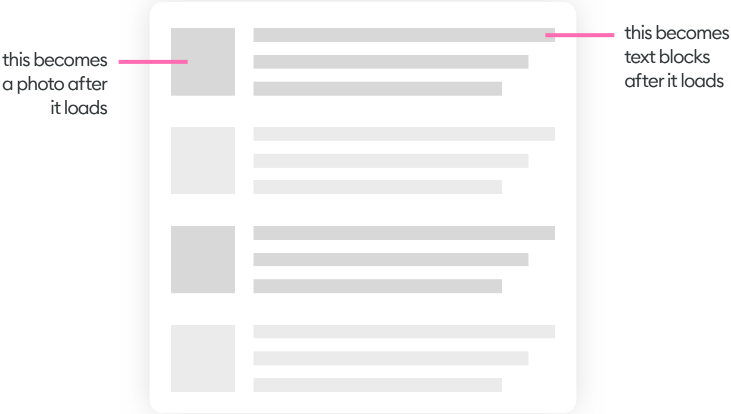
We are going to start with something you see when opening a freshly installed application. In the center of many apps, there is usually some kind of a feed or a list. What is the problem with that? Imagine you open a poorly designed clone of Twitter. In the beginning, you have no followers, so there is no content that can be presented to you. You are just staring at the blank screen until you find a profile of someone and start following them. The same goes for something like a todo app or any other list, that is meant to be filled with user-related or user-created data.

This blank state gets often lost in the process of design, simply because it's not exciting and it's not the state that the app is meant to be used in. Designers pack their interfaces with list views filled with perfectly designed cells to sell their vision to the client. Well-crafted designs should prepare for that, providing engaging content to guide the user and encourage them to take action. Or at least make it pleasant to look at.

Sometimes this fact goes unnoticed until it comes to the implementation, and someone launches the app with nothing in it for the first time. If you, as a developer are the first person to stumble upon that, you should bring this up and let your team know about it. Even better if you can notice that before, at the design project discussion stage.

> “Notice potential missing designs like empty states, validations and errors and ask your designer for help.”

The loading state is tightly coupled to the previous issue. If the list view is empty, it can either be so due to the lack of content or the content still fetching from the server. The luxury of a fast internet connection and a flagship device performance that many developers and designers get used to, especially in the developed countries, can lead to forgetting about problems like that. You need to always think about how the loading state. Users may try to use your app in different environments, including poor connectivity, resulting in a long wait before the content loads in your app.

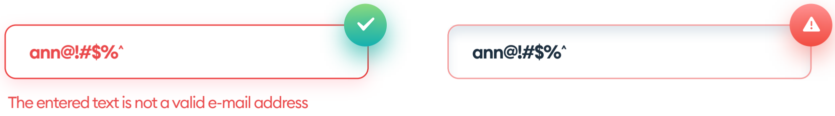

this becomes a photo after it loads
this becomes text blocks after it loads

Skeleton screens are one way of handling this problem.
You simply create coded "outlines" of a typical content with boxes or blcckframes where photos and text will load. You can also subtly animate them decreasing and increasing the opacity to show that they're loading.

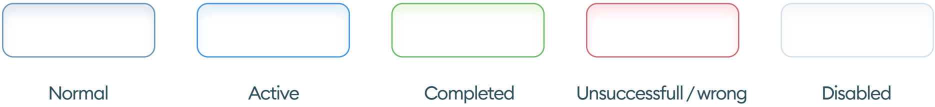
It would help if you considered that when it comes to displaying something on the screen during planning or implementing your app's interactions. Any action bound to the third-party service may not occur as fast as it is, when you are testing your app in your office with super-fast internet. Be prepared for that and plan accordingly by introducing something that will let your user know that the app is not broken, it's just loading.

FORM VALIDATION

If I had to point one thing that could make your users turn away from your app it would be forms. The longer they are, the more of a pain it is to fill them. They should be designed in a thoughtful way to make them as nonintrusive and pleasant to use as possible.


ann@!#$%^
The entered text is not a valid e-mail address
ann@!#$%^

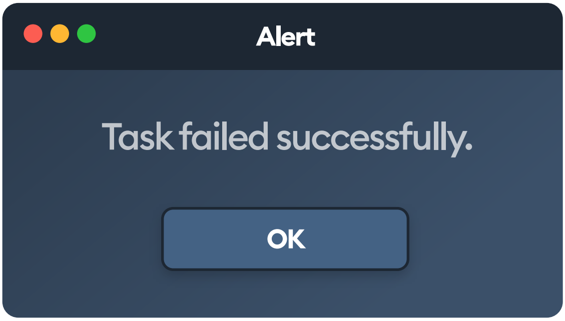
It's natural that a form can become a stopping point for your users when the provided data is invalid, and you need them to go through the process again. It's best, when they can clearly see what they did wrong. Generic error outputs, provided by your form validating library or a web service, that many developers turn to aren't helping. You need to be aware how the validation will make it easier for the user. Avoid just "adding some validation" and checking this problem as solved. You can also show the user when they filled a field the right way by highlighting the fact.



When dealing with form validation, you should talk to your designer. If they missed that step in the initial design, let them know and work together through all the fields and all the possible scenarios. Designers are usually thinking about how the app should look like when used correctly. They may also not know all the technicalities of your form inputs, hence needing your help to get this right. With a bit of help from the professional designer, you can prepare meaningful hints, presented in a friendly way, that can be even displayed before the user will press the submit button. All of this can make the mundane task of filling a form be a little easier on the user.

NOT DESIGNED

Designers don't always have all the intrinsic knowledge about your app's implementation details or how your authentication or backend works. As a result, some interactions may not be covered by the design project. You are probably better at identifying those situations. Speak up and let your team know about these, so they can be properly designed with the User Experience in mind.



Task failed successfully.

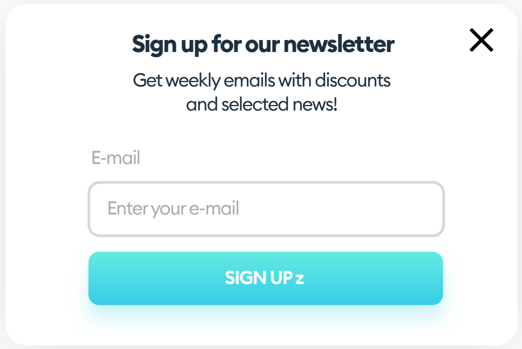
ALERTS & 404S

We know you test your software, don't you? Even if you are a "Super-Ultra-Senior-Developer™", you should always assume that something can go wrong. You can say that you are prepared for that because you implemented a function that will display an alert, when there is an error in your code or throw a 404 page when there is a server issue. That's good but not good enough.

There is a great chance that this is a generic view that you didn't put much thought into or used something built-in provided by the framework you use. Presenting a misaligned error page that looks off and does not maintain the other screens' design cues will look unprofessional.

NOT DESIGNED

Errors are one thing, but there are many other utility-oriented screens and messages that you need to present to a user in a regular app, be it a little toast message or a whole page. For example, confirming that the account has been verified or an invitation to the team has been successfully accepted. It's best if you don't have to figure out how to display it and can use a designed view.



Sign up for our newsletter

Get weekly emails with discounts
and selected news!

CODING EMAILS

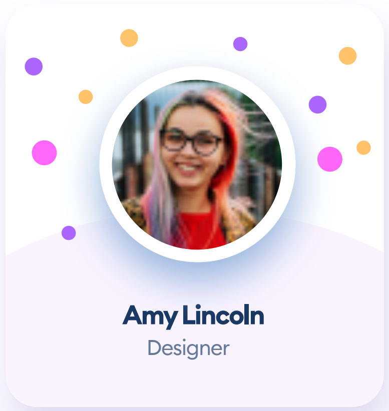
If there is an intent to send users any emails you are responsible for implementing, make sure there is a designed template you can use. It's even more important when you realize how many limitations you can encounter when trying to style the email or newsletter content. It's best if you could take part at this stage of the design process and make sure you'll be able to re-create the proposed layout later with code.

Work with your designer to figure out the best common ground between what looks good and what is technically possible.

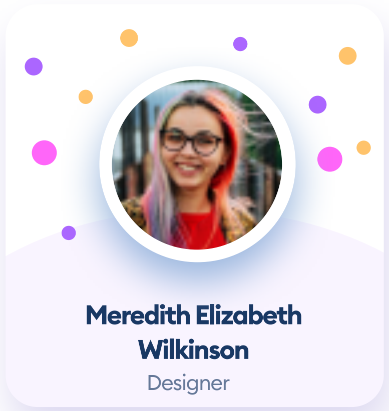
> “Emails are very difficult to code well due to a lot of limitations coming from email clients. ”

Design projects are made to look perfect because regularly, they serve more than one purpose. Apart from being a blueprint of how the product will look and feel, they are intended to make a good impression and "sell the project" to the client or a stakeholder. Designers may also want to share their projects on sites like Dribbble to show off their work. That's why everything is as polished as it can be.

When developers start to bring life to it by coding the screens and connecting real data, parts of the layout may start to feel broken. That's because those perfect elements that were adjusted to look good are often heavily influenced by their content. Real names and titles can be longer than provided examples. Now you can have a problem with not enough space. Something that was one line on the design is now taking two.

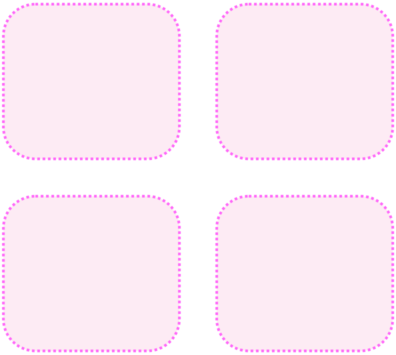


Design

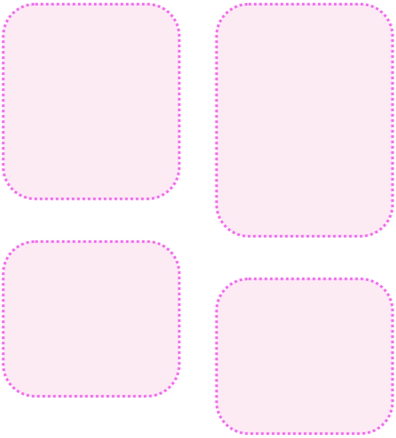


Coded version

As you can see in the example above, the designer assumed someone's name will fit in one line. But what happens when it doesn't? In the example on the right we push the subtitle down and the whole card looks badly aligned. It's best to plan ahead for scenarios like these.



Design



Coded version

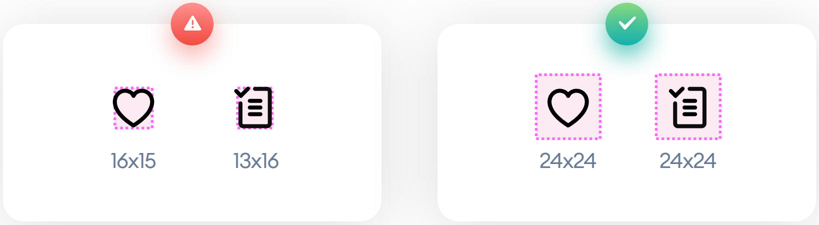
Components that form a grid or generally maintain the same height can be prone to such problems, too. Varying content length could be an end of their perfect alignment, now with the second element being too short and maybe the third one a bit too long. It's just an example, but you should be aware of those details and always check how your project changes depending on the provided content.

Most of the projects look good on high-resolution screens. You have enough space to fit the whole width of the layout without cramping anything. Problems arise when you try to see how it works on smaller devices like tablets and smartphones. If there is no design prepared for those breakpoints or at least a single mobile version, you may have to make your own adjustments to fit the content on the screen. You should at least consult your work results with a designer to ensure everything is readable and a good user experience has been preserved. You'll get better at this with time and each project completed, but at the beginning, it would be perfect if you could have a proper responsive design that covers all essential screen resolutions.

Turning a design project into a working app is a process that may go wrong on many steps, which may lead to a degraded quality of the final product. As a developer who cares about design, you should be mindful of that and make sure you are doing your best. Things like implementing the interface to look the way it should or optimizing the performance are critical to a well-executed development. It may require some effort from both you and the designer throughout the process, but it's worth it.

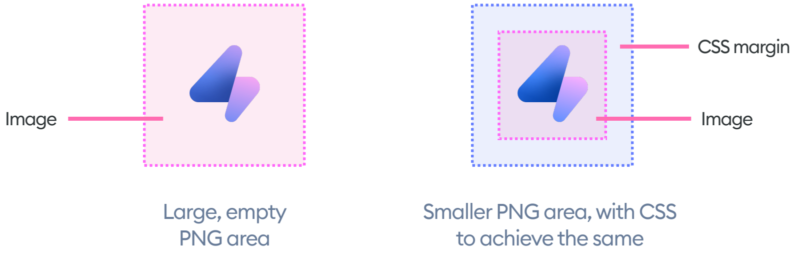
Handing over the design or assets from designers to developers has been significantly streamlined in recent years. With tools like Sketch + Zapier, Invision Studio, or Figma, developers can easily track changes in the design project and check all the spacings between layout components. Some of them allow for easy download of assets prepared by designers for each screen.

There are a few things you should remember about to make the implementation go as smoothly as possible. Often designers will use tools that automate asset exporting. Those tools aren't perfect and may produce assets that are hard to style correctly and may demand more customization in code.

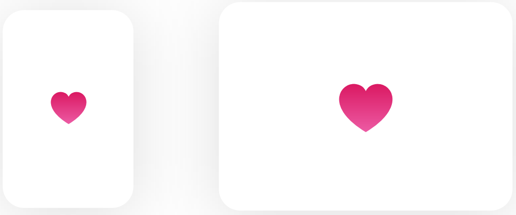


One example is cutting out icons without their bounding boxes - while it's easier for the designer, it's a lot harder for the developer to use assets done this way.

Shadows are a good example. Most of the time, you don't need an asset cut out with the shadow included. You are better off setting this yourself with CSS, which you can control and tweak easily.



Too much whitespace is another common issue. If you need some space between your image and text, you can use CSS to control the distance with more accurate values. If there is too much whitespace in your image, you won't be able to apply spacing measurements that you can read from your tool and position it yourself. You will also have to use negative values to make it smaller when working with responsive design on more compact screens.



Speaking of responsive design, when working with different screen sizes, it's a good practice to optimize your assets by displaying smaller files on smaller density screens. This will lead to a shorter load time and can speed up your website. Designers can export them in different multiplications of the original size (1x, 2x, 3x), depending on your needs or even in SVG.

> “Best products are made when both designers and developers understand each other and work towards the same goals.”

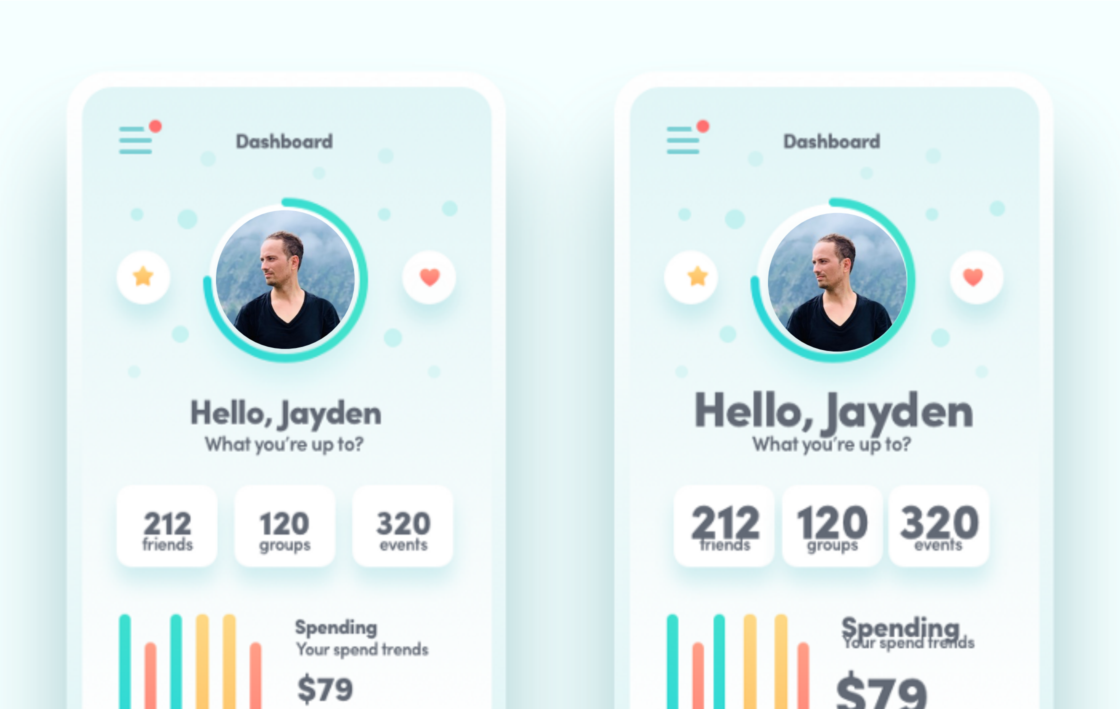
WORK TOGETHER

Good products don't come about without good design and good programming. The best products are made when both teams understand each other and work towards the same goals.

We hope that our few tips from this chapter can help you make the whole process easier. Streamline the cooperation by defining thought out requirements and providing support with your technical knowledge.

The result is a product that not just "works." It's easy to use because of its well-designed user experience and well-executed implementation, which does not get in the way of the design.

That way you'll end up with a well implemented screen like the one on the left, instead of the chaotic mess on the right.


Dashboard
Hello, Jayden
What you're up to?
212 friends
120 groups
320 events
Spending
Your spend trends
$79
Dashboard
Hello, Jayden
What you're up to?
212 friends
120 groups
320 events
Spending
Your spend trends
$79

Key takeaways

✓ Share your knowledge and insights with the designer. You may be able to improve the final experience of the product.

✓ Look at the design before you jump into code and start implementing the interface. If you can spot any future trouble, you'll make your and your team's life easier by providing feedback in the early stages.

✓ Don't always turn to your default or built-in solution when it comes to presenting any information. Well designed details make the difference.

✓ If you can, use dedicated software built for handing off the design to developers. You'll know what the exact hex code of all the colors is and what padding to apply to match the design pixel-perfect

7.

Semantics & Accessibility

Semantics & Accessibility

WHY DOES IT MATTER?

I bet you're not interested in semantics and accessibility. You want to build glorious apps, not delve into the nitty-gritty markup details.

I understand it. But you also need to understand one thing - semantics and accessibility are crucial for modern apps and may become a game-changer for your career.

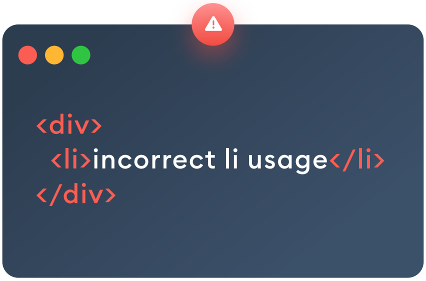
Accessibility and proper semantics serve similar purposes. They improve user experience and allow everyone to use the application - no matter their disabilities, age, and device used.

Everyone deserves to use your applications. You don't want to exclude people with disabilities. If not out of respect, then out of pragmatism - more users mean more money.

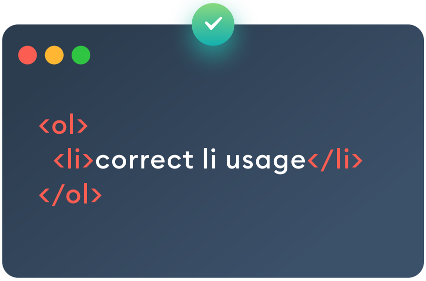
If you're still not convinced, I can assure you making semantic and accessible apps can boost your career. On the Internet, that's more and more dominated by Single Page Applications, developers care less about that aspect of good programming. It means you just need to know the basics to stand out. These basics can make a difference between getting and not getting a job.

So stay with me, it will be well worth it.

What is a semantic code, and why it matters? Semantic HTML is a meaningful HTML. Different HTML tags are designed to mean different things. When you use them correctly, your HTML is semantic.



```
<div>
  <li>incorrect li usage</li>
</div>
```



```
<ol>
  <li>correct li usage</li>
</ol>
```

There are about 100 HTML tags, and while you don't have to know all of them by heart, it's useful to take a look at the full list from time to time. We'll point out some of the most important in this chapter.

By taking care of the semantics, you help developers (future you included) read your code. You'll also serve users and clients a better product. Sometimes the visible result is exactly the same, yet it's still essential to use the semantically valid code.

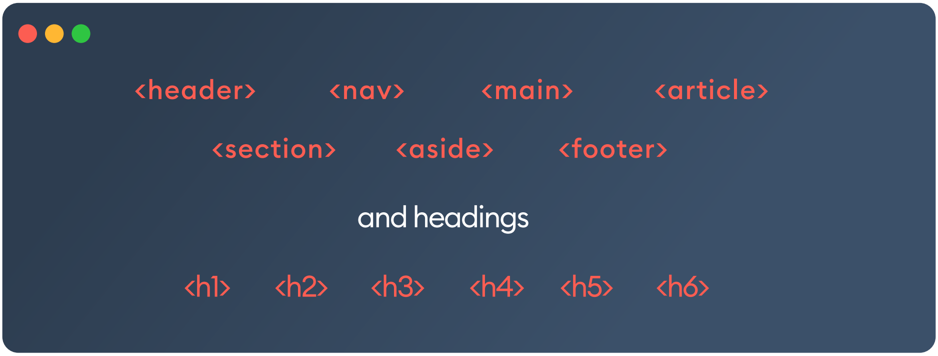
What value a proper markup provides to the user?

Reasonable markup improves your search ranking (SEO). And the better your search ranking, the easier it will be to find your page. Proper markup also improves accessibility, as it is used by screen readers to understand the page.

So semantic markup indirectly equals more users and more profit.

“Semantic HTML means you're using the right tags according to their purpose.”

Some of the HTML tags are the cornerstones that you'll use almost on every page. These include:



`<header>` `<nav>` `<main>` `<article>`

`<section>` `<aside>` `<footer>`

and headings

`<h1>` `<h2>` `<h3>` `<h4>` `<h5>` `<h6>`

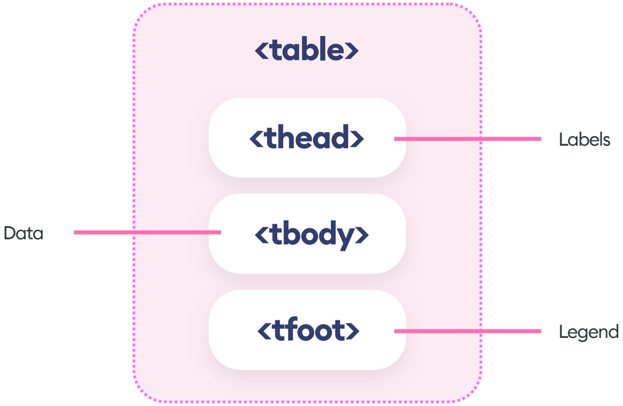
We're not going to elaborate on when you should use each of these tags, as you can easily find that information all around the Web.

What you need to remember, however, is if you don't have any of these tags in your HTML, you're probably doing something wrong.

<address> **<blockquote>** **<figure>** **<cite>**

<figcaption> **<code>** **<iframe>**

Some less frequently used tags will help you in more specific use cases. Check each of the above tags and what to use them for.



TABLES

Another thing to remember is to put tabular data in tables. In other cases, you make it almost impossible for screen readers to understand the data.

When using tables, remember not to put everything in the <table> tag. You should use <thead>, <tbody>, and <tfoot> for different parts of the content. You can treat a table as a separate page, with header, body, and footer, and use its tags accordingly.

FORMS

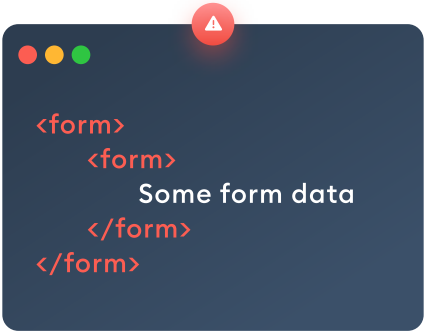
The forms are the main way for users to interact with your website. Making accessible forms is essential for website success.

To make accessible forms remember to use <fieldset> and <legend> tags when necessary. Additionally, make sure every <input>, <select>, and <textarea> have a corresponding <label> element. The label should be visible; never use placeholders instead of labels, even if you think it makes your UI look better.

ELEMENTS IN OTHER ELEMENTS

Another part of semantic HTML is knowledge of which tag can enclose other tags. Each element has rules on what are its possible children. It doesn’t make sense to list all those rules here, so here's a list of the most common errors.

Remember to never:
use <form> tag in a <form> tag,
use <anchor> tag in a <button> tag,
use <p> or <div> tag in a <p> tag,
use anything other than <li> tag directly in a / tag.



```
<form>
    <form>
        Some form data
    </form>
</form>
```

SUM UP

Choosing a proper tag for your use case is often more like an art than an exact science. It’s your job to consider each piece of information and choose the best HTML element available. Remember that <div> and <span> are the last resort. They don’t offer any semantic meaning; they are just generic spaces to put content. Every time you use one of these tags, reflect if any more appropriate tag does not exist.

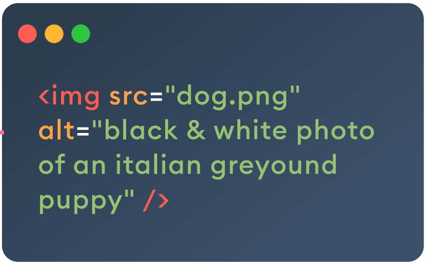
If you have doubts about how to use any HTML tag semantically, can check https://www.w3.org/. W3 creates HTML standards, and you can find every rule there.

> Some elements should not be nested within others. Don't make that mistake.

ACCESSIBILITY

Accessibility is a vast topic, and we'll only be able to scratch the surface. Our goal is to make the website the most accessible possible but still create it fast.

The first thing you need to know - there is an international standard for web accessibility - WCAG. Its current version is 2.1, and the 2.2 version is set to be published in 2021.



```
<img src="dog.png"
alt="black & white photo
of an italian greyound
puppy" />
```

ALT tag will help screen readers properly explain what the photo is.

THE BASICS

The <title> tag is a critical part of the page. It serves multiple purposes - it's shown in the title bar in most browsers, is used by search engines, and read by screen readers. Setting a proper <title> tag is crucial, as it will allow users with disabilities to know what the page is about. Make sure your title describes the content of the page and is different on every page of your site. If every page of your application has this same title the user can't decide which page is important to them.

From the Accessibility perspective, the second most important thing are alternative texts for images. More and more information on the Web is visual, so to help users with screen readers understand it we need to add alt property to the image.

Alt should briefly describe the meaning of the image. Meaning, not contents. For example - you should describe the icon of a magnifying glass in the search button as “search,” not as “magnifying glass.” If some image is just for decoration, you still should add the alt property, but set it to null, like alt="" (you should also add a role=”presentation” you can find more information about roles in the WAI Aria section).

Remember you don’t need to write “image” in the alternative text - the screen reader will let the user know it’s an image.

One small thing to include is the primary language of the page in the <html> tag. You set it by using lang property. You can also use that property on the other elements which are in different language than the primary language.

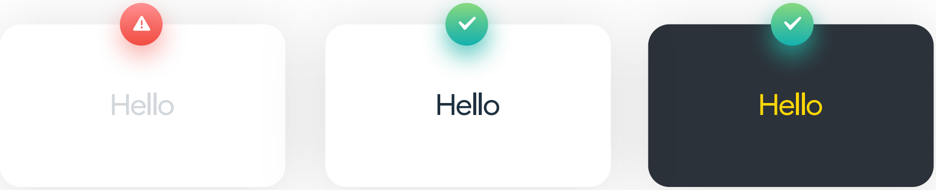
The next step to make the website accessible is making sure it has the right structure. We’ve discussed it in the Semantics section, so make sure you’re using correct headings and HTML5 tags.

You can use a small trick to emulate the experience of users with screen readers - disable all images on the page and turn off the CSS. This will show your page in plain HTML. If the page is still easily readable to you, that’s an indication of a proper website structure.

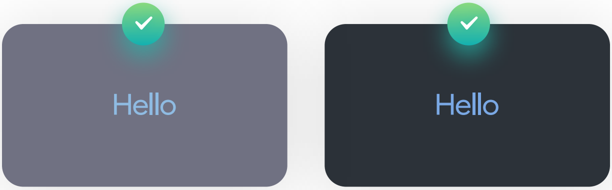
The next step is to show the page to someone else and see if they can use and navigate it. If they do, you’re set to go.

Not everyone uses screen readers. Some users are only mildly visually impaired and use your website in the standard way. To make a website accessible to them, you need to take care of a proper color contrast ratio.

The first rule is the contrast ratio has to be sufficient - for example, it is hard to read light gray text on a white background.

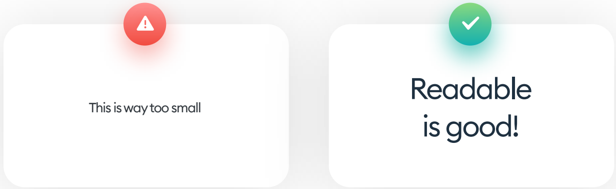


Also, for some people the bright colors are not readable; they need low luminance to be able to distinguish the figure from the background in a comfortable way.



Ideally, you should have two themes for your website - one with higher luminance and the other with the lower luminance. If that's not possible, you should aim for a standard ratio of at least 4.5:1. To check the contrast ratio of your text, you can use tools like contrast checker. Some designers also argue, that the website should be universally accessible with just one theme, even at the cost of reduced quality for most users. We'll leave that decision to you and your team.

The colors are not enough - you need to take care of the proper font sizing. The standard font-size for the body text should be at least 12pt (16px). Avoid font sizes below 9pt (12px), as they will be too small for many users.


This is way too small
Readable is good!

The great way to control the whole website's font sizes is by using the relative font sizes instead of absolute. You can use em, rem, or % values for every child of <body>. This way, you can change every font-size by just adjusting the <body> font size. You can even allow users to change the base font-size, thus providing them the experience they need.

No matter how good you'll set the colors and font sizes, they can be unreadable for some users. To handle that, you should make the text zoomable. Make sure the text is still readable when zoomed to 200%.

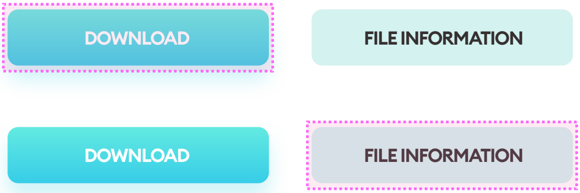
Some of the common problems with zoomable text to take care of are:
- the overlap of different sections,
- the necessity to scroll horizontally.

Make sure your layout avoids or handles these issues.

Making your text zoomable means not using images instead of the text. Images usually lose quality when zoomed, so the text becomes blurry and harder to read.

Forms are the major way users are interacting with your application, so it's crucial to make them as accessible as you can.

Remember that some people don't use a mouse at all and rely on the keyboard alone. Your page has to be also usable this way. To achieve it, you have to control the focus.



To see the focus in action click on some input on any website. You should see outline indicating the focus. In most browsers you can shift focus on your page using the Tab or Shift + Tab buttons on your keyboard. In the above example the focus is first on the Download button, but after pressing the TAB key, it shifts to File Information.

First of all, the focus should be visible. If you disable focus visibility, it becomes useless. The keyboard focus should also follow a logical order. It shouldn't jump from one place to another randomly, but go along the flow of the website.

The elements that can receive focus are <a> tags, form controls and <button> (not disabled). You can enable focus for other elements by assigning them the tabindex property.

Focus handling can become quite difficult so it's worth to test out your projects focus states before launch.

The most common problems are:
- some elements are inaccessible (it's typical with media player controls),
- keyboard trap - you can't focus out of some element,
- focus outline is not clearly visible,
- some functionality is not usable by keyboard only - like hover states
- dropdowns are not accessible using the keyboard,
- lack of focus on images that are links - make sure you can open the link using the keyboard (Enter button in most browsers).

We've already mentioned the importance of creating accessible forms and adding labels to all of your inputs. There is one thing to add - labels should be positioned correctly; for the languages read from left to right, the text labels should be on the left or top-left of the input, and radio/checkbox labels should be over the input.

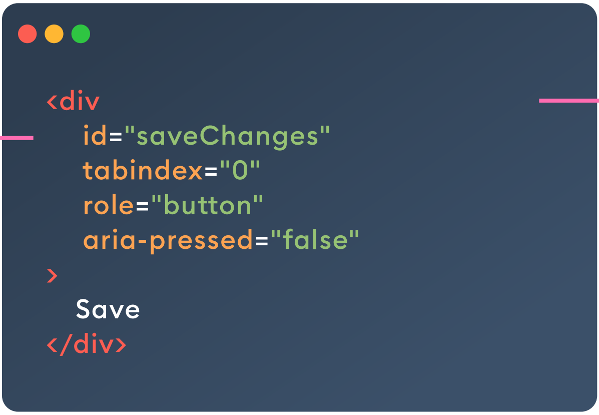
When making the forms, you usually have some special inputs. The most common one are required inputs. Take care to indicate them clearly - don't use color only indications, as some users are color blind. Put instructions to the inputs at the top of the form or the section they relate to. Also, if you expect input of some standardized format, place information about it in the text. If most of your inputs are required, it's best to simply show the non-required ones as optional.

Final thing about the forms - error handling. Make sure error messages give precise information about the error and the place it occurred. When a user gets the error the fields without errors should remain populated - the error in one field shouldn't clear all fields and make users retype correct information.

We've already established the importance of using semantically correct HTML. But sometimes it's just not possible. For example you can work with the legacy code where some important functionality or complex styling depends on the incorrect elements and you're stuck with the markup.

To handle that, we use Accessible Rich Internet Applications (WAI-ARIA).

With ARIA attributes you can make an element behave like another element. For example, you can use ARIA to let screen readers know that some <div> should be treated as a button.



```
<div
   id="saveChanges"
   tabindex="0"
   role="button"
   aria-pressed="false"
>
  Save
</div>
```

Consider this div a button

Make it focusable

The crucial thing here is ARIA attribute - role. It communicates that the <div> should be considered as a <button> from the accessibility perspective. That accessibility perspective is important - ARIA allows us to modify only the accessibility of elements.

It doesn't expand their behavior, for example, it won't make the <div> focusable. To make it focusable we added **tabindex** to it. As for the aria-pressed attribute we'll let you figure out what that means yourself.

“The first rule of ARIA is don’t use ARIA.”

Although you can use ARIA to change most HTML elements, it doesn't mean you should. Treat ARIA as a particular case, not as a standard in your code. There are many rules for using ARIA; you can find all of them **here**.

Some of the essentials that you need to know are:

- don't use if it's not necessary - semantically correct HTML elements are preferred,
- don't change native semantics unless you have to - try changing the roles only of <div> and <span> elements as they don't offer any semantic meaning,
- all interactive ARIA controls must be usable with the keyboard.

There are three main types of ARIA you can use:

roles - they define what an element is or does. Usually, they're similar to HTML5 elements like navigation or button. There are also additional roles as banner or search,

properties - they are like properties of HTML5 elements, for example, aria-required or aria-labelledby,

states - they are special properties that define the current state, for example, aria-disabled or aria-expanded,

You can find a great source of ARIA best practices at W3. If you want to test your skills, take a test at MDN. I think that you should test your skills as often as you can. It's a great way to learn.

Key takeaways

- ✓ being good in semantics and accessibility is a great way to stand out amongst developers
- ✓ semantic and accessible code allows you to reach more users and earn more money
- ✓ you have to understand what is the semantic meaning of each HTML element
- ✓ pay special attention to making forms accessible
- ✓ you can enhance the accessibility of the HTML element using WAI Aria

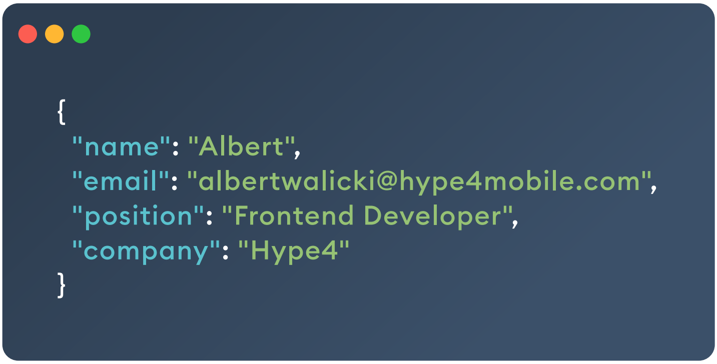
8.

Code is your business card

Code = Business Card

WHY DOES IT MATTER?

The only real indicator of how good you are at developing is the quality of your code and how it works. When you are looking for a job, many potential employers will ask you to show code fragments - no matter, if they're from your full-time job or from your side projects. This is one of the reasons why you should always keep it at its best. Your code is simply your business card.



```
{
  "name": "Albert",
  "email": "albertwalicki@hype4mobile.com",
  "position": "Frontend Developer",
  "company": "Hype4"
}
```

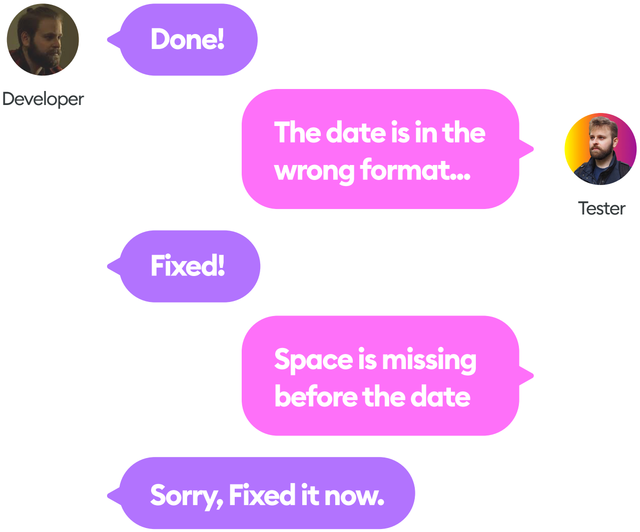
Your primary task in the work of a programmer is to understand the problem and business logic and write the code that will solve this task. In most cases, the code you write will not work from the very first moment. There are many potential reasons for this, for example:

- you did not handle request errors
- you forgot to pass, or you gave the wrong parameter to the function
- business logic required date in the format DD-MM-YYYY, and you returned YYYY-DD-MM
- you assigned a variable instead of comparing it (that's one of my favourite mistakes)

> “You are responsible for writing code that works.”

Lots of possible mistakes, right? And these are even not all of the errors that can happen. Therefore, it's essential to test the code you've written. I am not talking about writing tests in JavaScript (although this can work too), but about possibly the simplest form of a test - manually checking what happens when you pass various parameters to your function.

Checking it yourself and making sure that our code works saves everyone a lot of time. Clicking the ticket on the tester in Jira or Trello should be done only after making sure that everything is OK. There is nothing worse than wasting time bouncing the ball between developer and tester:



Done!

The date is in the wrong format...

Fixed!

Space is missing before the date

Sorry, Fixed it now.

Do you understand now, what the problem is? Conversations like these waste time and gets people unnecessarily nervous. The tester is also human, and in the barrage of tasks, they may not notice all the errors at once, which leads to a never-ending story of back-and-forth.

We can test logic in many ways; we have many types of tests, the most popular of them are:

Unit Testing – Software testing method by which individual units of source code, Intuitively, one can view a unit as the smallest testable part of an application.– *Wikipedia*

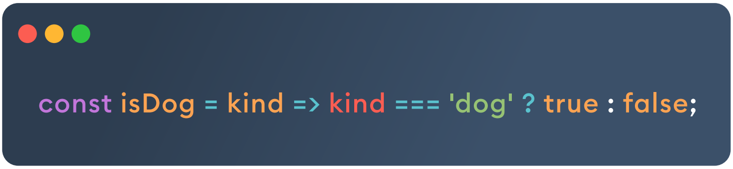
Functional Testing – Functional testing is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. – *Wikipedia*

Integration Testing – Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group. – *Wikipedia*

You do not need to know them yet, writing tests in javascript are not required when working as a junior developer. They often appear in the mid position, and in the senior position they are practically every time. In small businesses, you will usually not be able to write tests. They take a lot of time, and this time has to be paid by a customer who is not particularly interested in any tests.

So don't be too concerned about these tests. Just be aware that they exist. In this chapter, we will deal with the simplest form of testing - the already mentioned manual kind. This type of test will accompany you at all times. Many developers do not even treat it as a test, even though it is a form of a test in checking the correctness of the code.

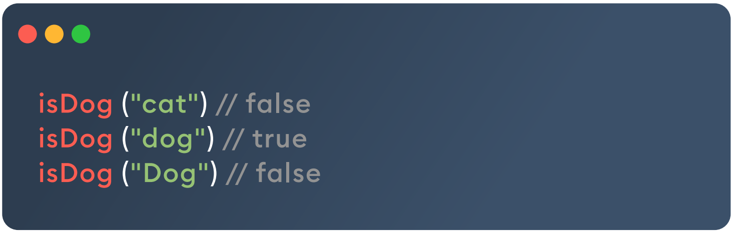
For example, let's check the following function together, it's very simple:



```
const isDog = kind => kind === 'dog' ? true : false;
```

The function is checking whether the given parameter that we get from the database is a dog or not. It is a straightforward function. At first glance, there is no chance of any error anywhere. Well, as an efficient tester of your functions, you immediately notice what can go wrong.

Let's call the function a few times:



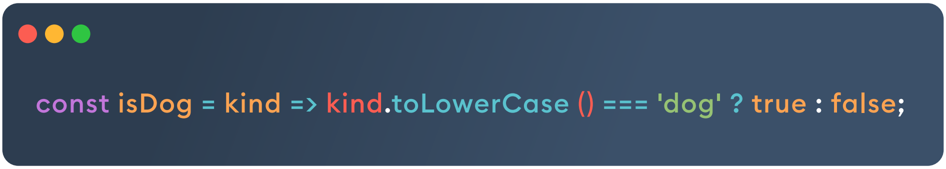
```
isDog ("cat") // false
isDog ("dog") // true
isDog ("Dog") // false
```

As you can see, the string 'dog' is not the same as 'Dog'. Therefore, the third time the function was called, we've got *false* instead of *true*. This error might happen if someone who completes the database, e.g. in a CMS, writes everything with caps lock turned on or as in our case - the first capital letter. That can cause our function to crash and return the wrong value.

This case is overly simplified, but such errors can happen. When the client completes the data himself, he may not know that he should enter a lowercase animal type there. Here also we have another problem:
- the frontend should use method to lowercase typed text, before sending it to the database?
- the backend should lowercase every text, which is user posting to it?

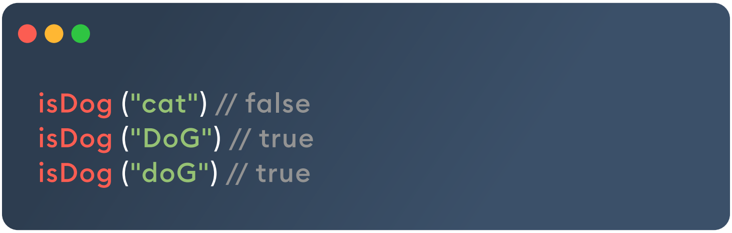
The answer to this question is: it depends on the project ;) We do not always have the option to change the backend, which is already ten years old, and we only do a redesign.

Coming back to our function, to fix it, just add the toLowerCase () method to the resulting parameter:



```
const isDog = kind => kind.toLowerCase () === 'dog' ? true : false;
```

Let's call our function again:



```
isDog ("cat") // false
isDog ("DoG") // true
isDog ("doG") // true
```

Now 100% each type of DoG will pass through the function. Of course, sometimes a different function may need to Capitalize, so this solution won't work in every possible scenario.

In all my years of working as a programmer, I've come across hundreds or thousands of problems related to just the layout.

Depending on the client and project, we are asked to support different browser versions. If we work for a corporation, unfortunately, there is a great chance that you will have to support Internet Explorer. Pray for it to be a minimum version of 11 ;)

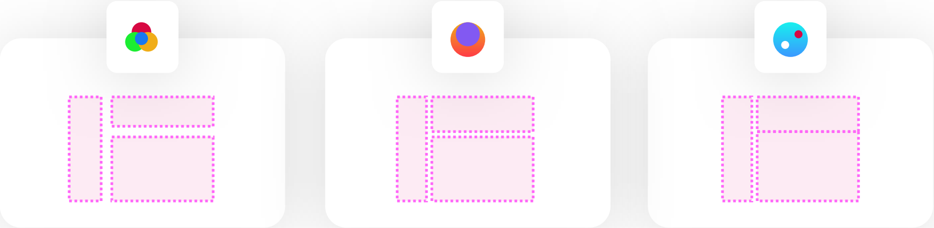
Fun fact: Back in the early 2000's all frontend developers dreaded having to do anything for Internet Explorer version 6. It has become so infamous in the industry for it's horrible layout rendering methods, that many companies created separate styles just for that one browser version. Sounds insane? It was!

BROWSERS

The projects you are working on or will be working on may differ significantly. Working on an application for a modern startup will not be the same as working for a financial institution or a government application. The latter have more stringent requirements when it comes to supporting older browsers, so as not to discriminate against people with older devices. It may be one of the reasons they need to "play it safe" in terms of the layout capabilities.

At the very start of a project, it's best to find out which browsers you need to support. If you don't have to worry about IE, you just saved yourself a lot of time! Many clients come prepared with a list of their own. Check that list thoroughly and try to negotiate a higher version whenever you see an old browser that may cause some problems.

Different browsers interpret and render items differently. Each browser has its problems; fortunately, there are not that many of them. Additionally, fonts render differently depending on the system. The most popular and supported browsers are:



There can be some small differences between how browsers show the same thing.
These are almost always possible to fix.

GOOGLE CHROME

Chrome has the largest share in the browser market. It reacts to bugs the fastest, and new versions are released very often. It supports the latest JavaScript and CSS standards that do not / work partially in other browsers. The most hassle-free browser available when it comes to development.

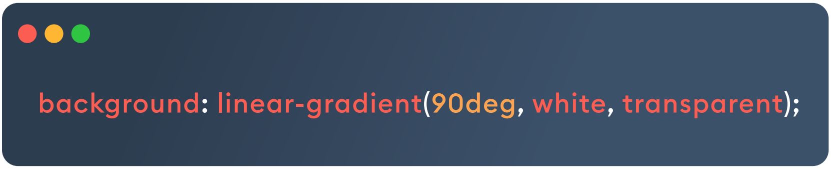
FIREFOX

Firefox, just like Chrome, is continuously releasing new versions of browsers, fixing bugs and releasing new features. Unfortunately, this browser has one major drawback when it comes to the frontend - it renders fonts much thicker than other browsers. If we want a nice font to be rendered here too, we have to compromise with other browsers.
You can read more about fonts **here**.

SAFARI

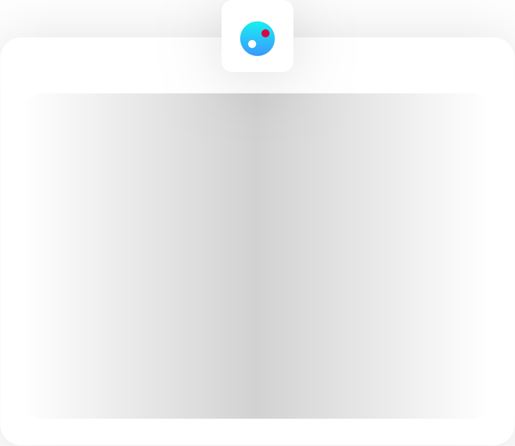
Testing on safari is cumbersome mainly due to the lack of access to this browser from Linux and Windows operating systems.
Yes, to test anything on safari, we need a MacBook, iPad or iPhone.

My favourite "problem" is the opposite of transparent "color" compared to all the other browsers. It's easiest to spot by using a linear gradient.



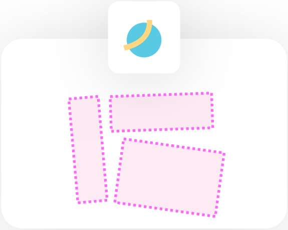
```
background: linear-gradient(90deg, white, transparent);
```

We expect the gradient to start from white to “no color”.
Unfortunately, as you can see in the picture below, Safari works a bit differently.



The easiest way to solve it is to replace 'transparent' with 'rgba (255,255,255, 0.001) - white color with opacity 0.001.

Internet Explorer is an old browser used on computers with the Windows operating system. Very often, it needs to be supported in financial or corporate projects, because it's pre-installed on many corporate machines as a default browser.



Many modern solutions do not work well with Internet Explorer. Some of the examples include:

- display grid (you have to write some fixes)
- no support for min-height / min-width
- position sticky
- clip-path
- fit content
- ES6

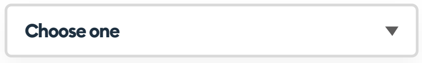
And many, many others.

Microsoft Edge is the successor to IE, which fortunately keeps getting better and better, making it a lot easier to develop for than IE.

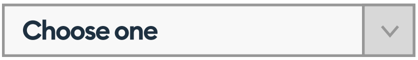
COMMON PROBLEMS

There are some problems that are common to all the browsers.

One of them is different interpretations of inputs and selects - different default settings that need to be hidden when using custom inputs / selects, so they can look the same in every browser. It's always a choice between OS uniformity (that the users already know) and your system uniformity, as the users can use it on a variety of devices. In most cases we believe it should look the same on all devices, even at a cost of custom styling all the OS enforced controls.

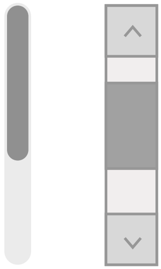


Custom select



Old-style "default" select

Scrollbars - in each browser we have different thickness and different appearance of scrollbars. When we want to create, a modal that scrolls, the default is usually an ugly, system-style scrollbar. That can completely break the carefully crafted UI with an eye-sore element that doesn't match.



Not all browsers (e.g. IE, older safari versions) are supporting new image formats like WebP. You can check which browsers support what at: **https://caniuse.com/#feat=webp**

> Many browsers - many different problems ”

Before you start building responsiveness on your website or application, you should find out what breakpoints you should use. Every company is different, but I have generic breakpoints for you:
- Desktop - Above 1024px (inclusive), this is the horizontal width of the iPad
- Tablet - below 1024px up to 650px
- Phone - below 650px

Of course, that doesn't mean I'm only using these three media queries. Each project should be checked every few pixels, preferably from 2560px (2k resolution) down to 320px (iPhone 5).

Desktop (px)

2560	1920	1600	1536	1440	1366	1280	1024

Tablet (px)

1080	1024	800	768

Phone (px)

414	375	360	320

In many cases, such as carousels on mobile devices, more custom breakpoints need to be added. With the mentioned carousel, you have to add media query often - even every 10-20px to keep the assumed design.

Many programmers use dev tools in Chrome, Firefox or other development tools to test the mobile UI. This is roughly correct and sufficient, but I always recommend checking how the application or website will behave on your device. The simulator built into the browser reflects only the width and height of the screen. There are no "touch events" there, scripts checking what devices we use will still detect the desktop version

Most of the products that exist have lousy design or UX. Usually, it is the fault of the designer, the owner of the product (it should be the way I want it because I know best what if it breaks all good patterns and is even anti-pattern), but it can also be the fault of a developer. I met with projects that had a beautiful design, and the coded app was tragically awful. It was not even similar to the delivered project. In the worst of such cases, the developer asked our designers for the PNG version of the entire project, so that he could insert it as a background-image and he would add some absolutely positioned buttons.

Nightmare.

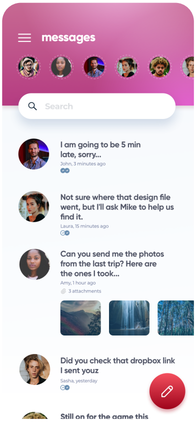
Fortunately, the client did not agree to something like that, and the entire frontend passed into our hands.

I don't like working with projects that don't make business sense to me, have poor design and even worse UX. Thanks to our partner and founder of Hype4 - Michał - I am more sensitive to such things. Thanks to this, I can help a client how to solve the problem in their application, which significantly speeds up the entire process.

As a frontend developer, you must be able to perfectly copy the project into code, because this is your most essential function in the company. Each of us started from learning HTML and CSS. These two things are the easiest to master. Of course, I am not talking about miracles like CSS images (works only in chrome).

All projects that we create in our company have to fit 95% of the design. This may seem very difficult to reproduce but is not the case if you are using the right tools. I'm using the PerfectPixel tool for Google Chrome, and I recommend it to you as well. This extension allows you to apply a design on your website.

This tool flawlessly shows us how good or how bad the project turned out to be. Let's take a look at the picture:



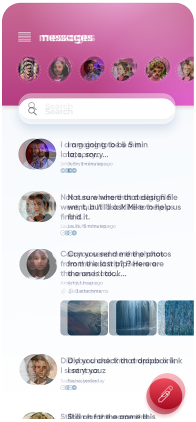
I am going to be 5 min late, sorry...
John, 3 minutes ago

Not sure where that design file went, but I'll ask Mike to help us find it.
Laura, 15 minutes ago

Can you send me the photos from the last trip? Here are the ones I took...
Amy, 1 hour ago
3 attachments

Did you check that dropbox link I sent youz
Sasha, yesterday

Still on for the game this



In both examples, we have the same design superimposed on the coded simple view. Do you see the difference? Padding around too small, the text is not in the centre, other text is not aligned with the title. This can be corrected very quickly. On the right side, the project overlaid on the coded view merges into one, it's hard to say what is what. This is the accuracy we want!

Of course, you can't always be that precise. If someone who provided you design did not contribute to it as much as should, it is crooked already at the design stage. In this case, we should throw out pixel-perfect from our head and code the view according to our intuition and a few rules from the Design Basics chapter.

Key takeaways

- ✓ You are responsible for the quality of your code. Keep it at your best
- ✓ There are multiple different tests. The easiest one is a manual test, which will accompany you every day
- ✓ Always test your websites and app in all common browsers and screen resolutions
- ✓ Use PerfectPixel extension to convert the design to the almost perfect layout

9.

Back-end basics

Back-end basics

WAIT! WHAT?

A chapter about backend in a book dedicated to frontend developers? That's right, it's not like we switched to a Backend Unicorn book in the meantime. After all, if you are building a frontend app, there is a great chance that apart from the website content you put there, you have to communicate with some kind of a backend service.

We think that a sprinkle of backend knowledge will not hurt you and might be beneficial for you and your career as a fulfilled developer.

Don't worry, we are not going to dive deep into the server configurations, querying the database, or writing custom middleware. That's not the case here. You are a Frontend Developer after all. We are going to stick with it.

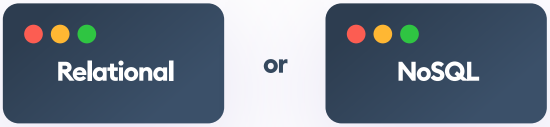
What we would like to achieve here is for you to be confident enough in this subject to speak about backend infrastructure from the perspective of a frontend app with other developers or your clients. That will improve communication, which is one of the most important parts of success in any industry.

If you want to be able to talk to the backend developers, you have to know how your app can talk to the backend service. That's why you should know what an API is, and how you can use one, what types of databases there are, how your app is deployed and served to the public. Understanding the basics of backend architecture and its tech stack will make you a well-rounded programmer that can understand the bigger picture of how the product works.

Starting with the basics. Say you want to save some information about your users, e.g., their first name and date of birth or any other data, they will access later. **Hey, I can use local storage for that, or even some cookies**, you may say. And that's true. Perhaps, you just solved the problem we faced and don't need a database for now.

What if I would like you to make those pieces of information persist for a bit longer, like a day or two or even a week? Ah, you're right, your solution is still capable of that, you can make the cookie persist for longer.

But what happens when the same user tries to access your app from a different web browser or their mobile phone? They will be presented with a clean slate with no access to any of the data they left in your system. In most cases, you need an account for that. To keep track of your users and let them access their accounts from different devices, you need some kind of a database. Be it your own hosted solution or a service that you can connect to.



What database do you need?

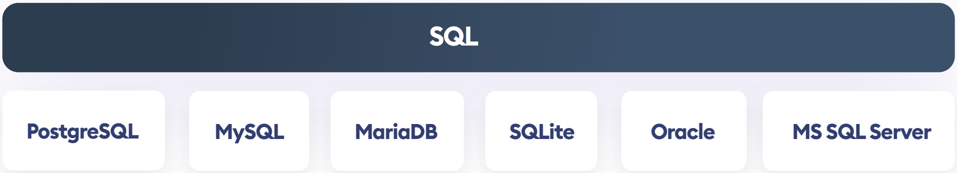
There are two types that we will cover in this book. We know that those are not the only ones out there in the wild. But as a Frontend Developer, you will be working almost exclusively with those two types most of the time:
- Relational database
- NoSQL database

The most common and well established type of database. Relational Database Management Systems (**RDBMS**) are with us since the 70s. With systems existing for so long comes a great advantage of them becoming an industry standard. If they weren't, they wouldn't be maintained or developed now.

But they are, and many experts work on them, introducing new features and taking care of keeping them at the top performance.

PostgreSQL, MySQL, MariaDB, SQLite, Oracle, Microsoft SQL Server - all of them fall into this category. You may have heard these names before, as they are some of the most popular solutions when it comes to Relational Database Management Systems.

To operate such a database, you use a Structured Query Language (SQL) - hence their popular name: **SQL Databases**. With those, you structure your data by splitting it into different tables or entities that can be related to each other. Tables maintain a rigid structure with columns defining properties. Columns can implement many various constraints and can serve different roles to make querying more performant.



All the databases mentioned above have to comply with the SQL standard, which can be a significant advantage for you. It means that if you know one of them, you should be fine switching to another one without having to re-learn everything (at least when it comes to the basic functionality). They'll differ in terms of any additional features or how they are implemented, but the principles remain unchanged.

Newer type of database with a wide spectrum of possible applications. As the name implies, you don't use SQL to work with these databases. It doesn't mean that they don't use their own implementation of a query language. They do, but it's not a standardized language that you can use with other databases. Often they are modeled after SQL, but that's not the rule. Some of the NoSQL databases are even simple enough that you don't need one to query your data.

NoSQL is a broad term, and there are many subtypes of databases included. The way they work is not as uniform as with relational databases, where every solution implements a defined standard, adding their own functionality on top of that. Here, it's important to keep in mind that NoSQL databases can be designed to solve a particular problem that may not fit your use case. You have to be more cautious about choosing one.

NoSQL

MongoDB
Redis
Amazon DynamoDB
Firebase Firestore
Cassandra

MongoDB, Redis, Amazon DynamoDB, Firebase Firestore or Cassandra, are a few products you will most probably hear about as a frontend developer.

With a strong emphasis on the first one. Why this one? First of all, it became a sort of a go-to database solution for backend services written in NodeJS. It's also very common for other backend stacks to turn to MongoDB when searching for a nonrelational database.

> “SQL based databases are the most popular kind, that you'll likely be working with the most.”

In most cases, your frontend app will not communicate with the database directly, not if it's a remote database stored on a server. So, as it's living there on a server, you need some code living on a server as well, that will do the talking part. In other words, you need a backend. It can come in a variety of flavors, but it boils down to two approaches. You either use something called an Application Programming Interface, **API** in short, or you build your frontend as a part of a bigger backend app that serves your HTML pages.

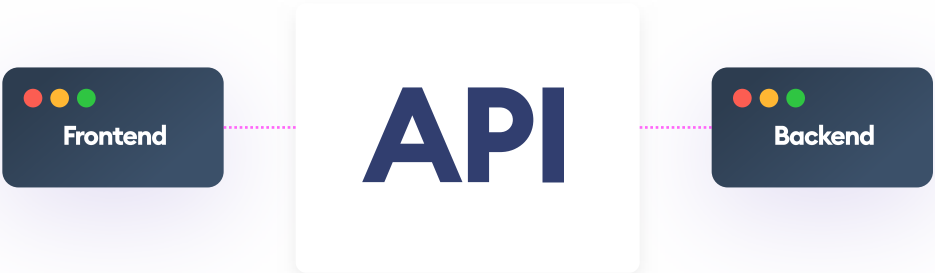
With the latter, the backend framework takes care of rendering your HTML that you write in so called **templates**. Templates are created with **template languages** that can be more or less similar to the HTML structure you are already familiar with. Their most important feature is being able to use **placeholders** and macros to insert data fetched by the backend into your website.

This way, you don't have to interact with the database yourself, you only use what has been provided to you inside the template. The same goes for sending data to the database. You write your usual forms in the template, with the designated action, and the backend takes care of processing the values submitted through your inputs.

We're not going to cover this scenario any deeper because the implementation can differ between template engines and backend frameworks. It would take the rest of the book to explain how all of them work. However, the main idea is similar for all of them, and you just stick to writing your HTML, CSS, and JavaScript.

There is much more you need to learn, when you want to create a Single Page App with a frontend framework like React, Vue or Angular and using **APIs** to get your data. First, let's briefly cover what an API is.

Application Programming Interfaces are there to let your app speak with another codebase via a standardized manner and make use of their functionality and features. APIs abstract all the complexities and let you use simpler syntax.



So in simple terms, API is what helps you easier connect your frontend to the backend.

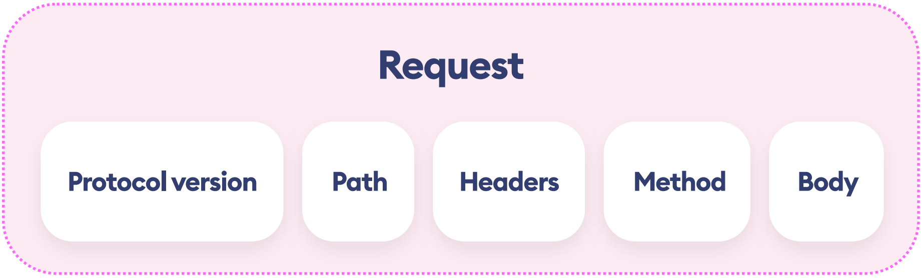
You've probably used some APIs already if you did a little bit of a fronted programming. When learning JavaScript, you probably used DOM API to manipulate your website's content, or maybe you called a `setTimeout` method. You didn't have to interfere with any complex time measuring internals and simply called a function. That's what an API is for.

We will be talking mostly about other APIs, that are not built into your browser or operating system. We are going to use third-party APIs, that are provided by external services.

Most of the APIs you are going to see in modern web development come in two flavors - either a REST API or GraphQL. Although they work differently, you write your queries using a different syntax, and you probably use distinct tooling, at the core, you will interact with them using the same tool. Because for all of them, you can use a simple HTTP Request.

Requests are a type of messages you send through the HTTP protocol. They have a defined structure that decides on an action you want to perform using the Request. With Web APIs, you work on a client-server basis. Client (your Web app or Mobile app) sends requests to the server, and the server sends back a Response. Your browser is constantly doing Request, when you open any website. Actually, opening one is a request in itself.

Request consist of protocol version, path (with optional query parameters), headers, method and body.



You don't have to declare a **protocol version** that you want to use in most cases. Just for a reference, those are the values you may see being passed at this parameter: HTTP/1.0, HTTP/1.1 or HTTP/2.0.

Path is going to be a URL of the resource you want to access on the API server. It may be a full URL like so: "https://yourserver.com/users" or maybe stripped down to a meaningful part "/users", when your frontend app is hosted on the same server as the API.

Have you ever thought about those long random-looking strings when using different websites? Those are **Query Parameters**. They often look like gibberish because they are encoded, but they are important when you want to pass some additional parameters to the query you make with a request.

> API is a way for backend and frontend to talk to each other, even if they're very different technologicaly. ”

The purpose of **Headers** is to pass some Meta information about the Request you are doing, about the resource, or about the client doing the Request. Using "Content-Type" header with a value of "application/JSON", will notify the API that you are sending some JSON data in the Request. You will often see headers being used to authenticate users or your app bypassing Authorization Tokens.

Now let's look at **HTTP Methods** that you can use with your requests. There are only a few of them, and each one serves a different (sometimes similar) purpose.

When interacting with API resources, you'll be using:
- **GET** - to retrieve some data of the specified resource
- **POST** - to submit an entity of the specified resource
- **DELETE** - to delete the resource
- **PUT** - to replace the resource with new data you submit
- **PATCH** - to partially modify the resource

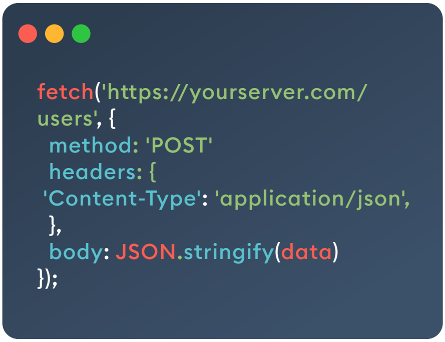
Yeah, those names are pretty self-explanatory, but it's good to know what you are supposed to be doing with each method. I've seen some people use PUT instead of PATCH or vice versa because both look like they are doing an update to the resource.

Body is an optional part of the Request, but you will not create much without it. All the data you want to send to the server, for example, values submitted by a form on your websites, have to be sent in the "body" parameter. For the most part, when working with web APIs, you will be sending a JSON there.

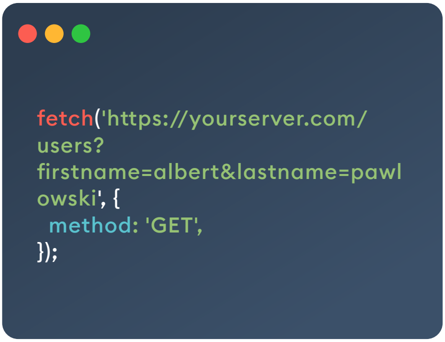
With all that knowledge, you should be ready to send your first Request.

In JavaScript, you can use `fetch()` to send requests to the server. With this function, you pass the path as a first parameter and an object of request options as the second parameter.

Here is how a simple fetch call can look like. This example could be used to create a user with provided `data`. We point to the `/users` resource on the "https://yourserver.com" server. "POST" method means that we want to create a new entry with data provided in the`body`. We set a header of "Content-Type': 'application/json' so the API knows that we are sending a JSON object in the body.



```
fetch('https://yourserver.com/
users', {
  method: 'POST'
  headers: {
 'Content-Type': 'application/json',
  },
  body: JSON.stringify(data)
});
```



```
fetch('https://yourserver.com/
users?
firstname=albert&lastname=pawl
owski', {
  method: 'GET',
});
```

Let's also look at another example request. The one on the right should return all users with a name of "albert pawlowski"

Our path points to the /users resource on the "https://yourserver.com" server. 'GET' in the method field means that we want to retrieve some data and "?firstname=albert&lastname=pawlowski" are query parameters that API can use to filter the users resource and return all users with a first name of "albert" and last name of "pawlowski".

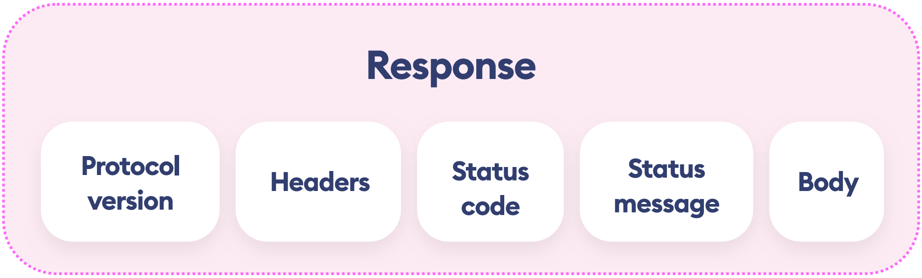
That's one way of interacting with an API. It's not that scary now, isn't it? Instead of `fetch()` you can also use something like `XMLHttpRequest` or a third party library of your choice like Axios.

Ok, we sent the Request. What about the Response? Isn't that why we are doing all of that?

Sure, let's see how it works.

Just like with requests, responses are a type of HTTP messages that have a defined structure. It's even made up of similar building blocks.

A response consists of the protocol version, headers, status code, status message and body.



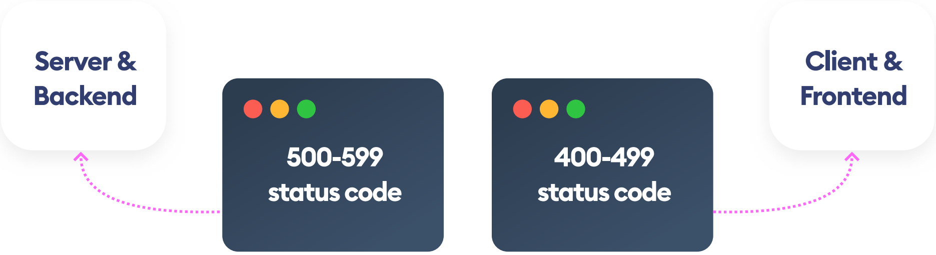
You've already seen a big part of that when we discussed requests. Response headers work in the same way as request headers, but now sending information about the server instead of the client.

Status code is the new thing. We don't have those in requests. This code indicates whether our Request has been processed successfully. You may have already heard about the famous 404 code, which means that the server didn't found the resource that Request was asking for, hence the usual message - "Not found".

> “Working with an API you send requests to receive a response.”

You can look up all the other status codes in the HTTP documentation, but here is a brief summary of status codes you may encounter:

- 500 - 599 - server errors (backend developers did something wrong)
- 400 - 499 - client error (you did something wrong)
- 300 - 399 - it's a redirect
- 200 - 299 - your Request was successful!
- 100 - 199 - represents an informational response



While looking into a potential problem, keep in mind that the 500's are for backend developers, and 400's are for frontend.

That's exactly the role of the status message - it's a short description of the status code in the Response.

Response body is optional, just like with requests. If you made a "GET" request, you'll probably see your data being returned there.

That's all you need to start working with web APIs. Requests and Response are going to be your bread and butter when doing so. Now you are prepared. Go and build something with some fancy APIs!

Or maybe wait a minute, and let's look at the example stacks that you could be working with when we speak about backend technologies.

First, let's understand what types of architectures are being used when we speak about the web apps' backend and frontend.

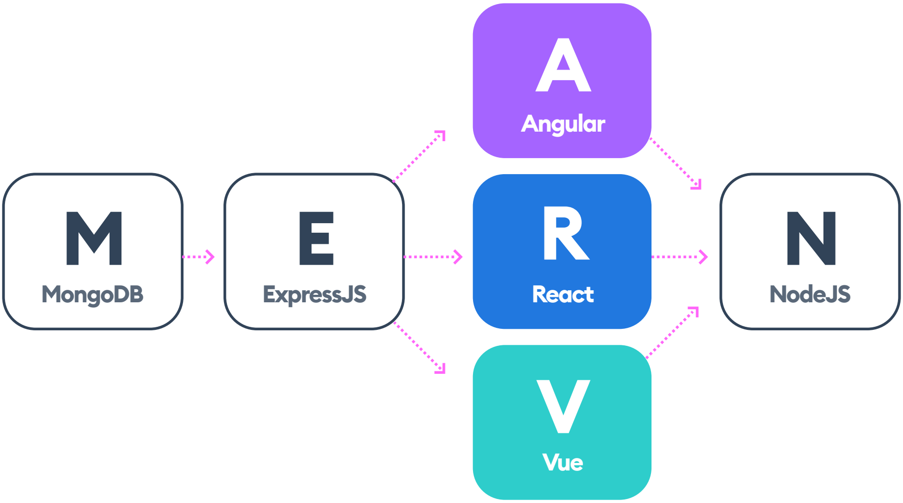
The most simple is the one we briefly mentioned before diving deeper into the API world - simply using a backend framework with a template engine. You don't have to use API when working with them, but you may need to do some requests nevertheless. All your work is mostly around HTML files / templates, CSS, and JS, and the backend is responsible for providing all the data in templates, ready to be used by your code.

Popular examples and programming languages they use:
- ExpressJS (JavaScript)
- Django or Flask (Python)
- Laravel or Symphony (PHP)
- Ruby on Rails (Ruby)
- Spring (Java)
- ASP .NET Core (C#)

Nowadays, you may see a different approach to building web apps being popular. It's about building a Single Page App with a frontend framework of choice, e.g. React, Vue, Angular, Svelte (and many other) as well as building an API on the backend side, that is going to be utilized by the frontend app.

The API can be built with any of the backend frameworks mentioned above. For you - a frontend developer - it doesn't really matter which one it was exactly. You will use a method we learned in this chapter to perform your requests and never touch the backend code.

You may be working with a Django backend or Rails or .NET and don't even know about that. You may even don't know what type of database is being used (you would probably get to know if it's SQL or NoSQL). But there is one special tech stack worth mentioning for JavaScript developers. It can be a variation of MEAN / MERN / MEVN.



NodeJS is the JavaScript environment that backend code is running on, and ExpressJS is a framework that streamlines the process of building an application in that environment.

It's popular among frontend developers because it's using JavaScript language both on the frontend and the backend side. It's supposed to simplify communication and interoperability between developers as they don't have to switch context that much when they want to do some work on the other side.

This is also a good starting point for experienced JavaScript developers to create their first API and maybe build a simple, full-stack web app.

Key takeaways

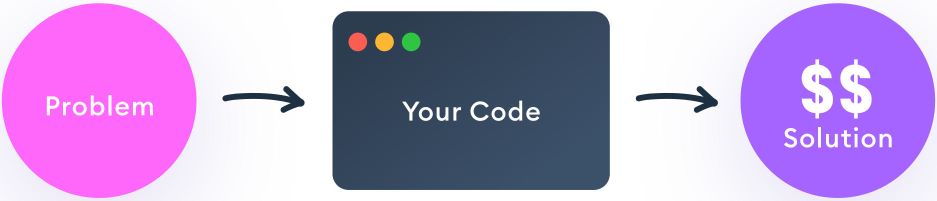
- ✓ If you need to save some data that has to be tied to your users - you need a database that "lives" on the server
- ✓ Relational databases are a perfect fit for related data with a defined structure that you can split into many tables.
- ✓ SQL is a query language that is a common standard for relational databases. If you learn it once, you will be to apply that to many different database engines.
- ✓ NoSQL database is the right choice if you are going to work with unstructured data or you need to make frequent changes to your data model.
- ✓ APIs make it easier for you to interact with other codebases systems and apps.
- ✓ In web development, most of the time, it's going to be in the form of a REST API or GraphQL API.

10.

Your code is not the product

Code vs Business

YOUR CODE IS NOT THE PRODUCT



That’s the single best career advice I ever received.

I’ve fallen in love with programming because it can be so elegant. The great programmers can create code that is so clean and readable that you can even call it beautiful. I, for sure, often call it this way.

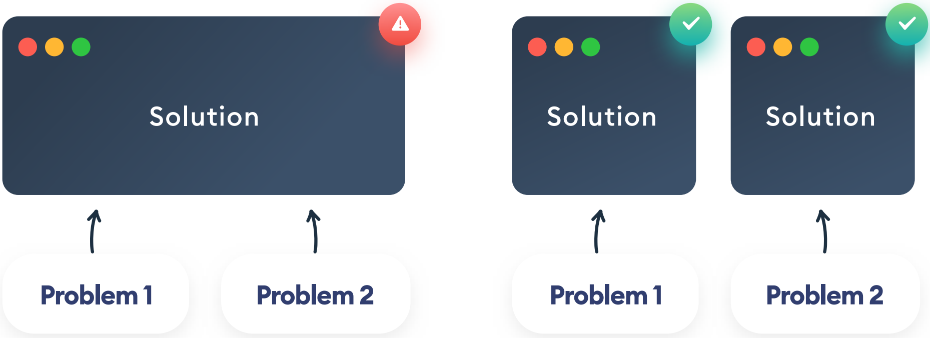
Some solutions to algorithmic problems blew my mind with the brilliance of their creators. Some codebases were so robust - performant, maintainable, and expendable - that I instantly craved to work with them.

Excellent code is the aspiration of many programmers, and it should be this way. But perfect code isn’t always the best choice. Sometimes, and by sometimes I mean quite often, you don’t need it. Sometimes striving for perfection is simply a waste of time.

It may sound like blasphemy but stay with me.

Real-life programming is not the contest for the best code. It's about solving real problems and creating products. It's about providing value.

To provide value for your clients, you have to understand their needs. You should know what the product's business goal is and make decisions based on that.



You shouldn't try to fit a single solution to every type of a problem or product.

Instead choose the most optimal approach that's created with the problem in mind.

Is your client a startup that wants to build MVP fast, has a tight budget, and only half a chance of succeeding big? Or maybe it's a huge bank that needs secure and reliable software for the decades to come? What are the essential features of the app? Is this app to be used by thousands of users in the next few years or just a few hundreds of users at the one-day conference?

You have to know the answers to those questions if you want to offer viable solutions. A considerable part of your job is recommending appropriate solutions to solve client's problems. Perfect code is rarely the answer - good enough code that does what's required is.

To become a great developer, you need to resign from code-oriented thinking and embrace solution-oriented thinking.

You need to be wary of the project's requirements, even if the client isn't. You are the expert that can and should advise what the possible solutions and problems are. It's the norm that the tasks aren't defined because the client doesn't have the expertise. If they had it, they wouldn't need you.

Communication with the client in that regard is one of your crucial duties.

Learn how to communicate with clients, but better yet - how you can understand each other. Avoid speaking 'in code' to non-technical clients. Find a common ground and show them you're there to solve their problems fast and efficiently.

You may think that you don't contact the client. Some seniors do it, you just do what you're told. Maybe. But your goal is to grow and become an essential part of the team. And that will involve communication with the clients at some stage. So be sure you'll become good at that.

Understanding the potential client's priorities makes a difference between getting a project and being rejected. Both managers and clients covet developers aware of business goals and limitations.

“Good communication with the clients makes you indispensable as a developer.”

It doesn't matter how good your code is.

Or rather it does matter. Providing you understand the requirements and priorities. To provide value, not just code, you have to be productive. Productive, not necessarily busy or hardworking. Productive people do things that really need to be done.

Programmers have their interests and priorities. We waste countless hours on non-essential things. We spend days on improving the performance of code that's already performant enough. We polish our CSS as long as it's pixel-perfect at every screen size. That's great. Of course, if it's also our client's priority, not only ours.

Programmers know that our job is all about the tradeoffs. There are lots of them, like speed vs. ease of use, low run-time cost vs. ease of debugging, simplicity vs. control, etc.

But we often forget that building things for clients is also about tradeoffs and priorities. **And it's not our priorities that matter. It's theirs.**

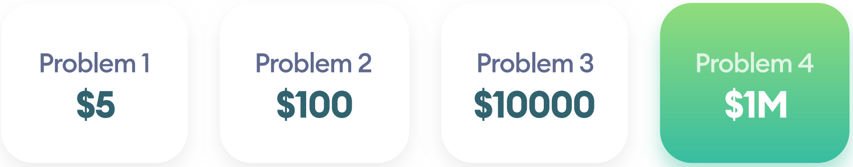
Think about it from the client's perspective. Imagine your team is responsible for creating the conference app front end – attendees will use it to review speakers and send their feedback. The app will be used only once – for the conference. It's a simple project with a weak deadline.

Which developer does the client prefer? The one who took four days to write maintainable code responsible for searching the speaker and a single day for other tasks, or the one who talked with the client about the priorities and focused on an intuitive user interface?

You guessed it. The client always chooses his needs and priorities over yours.

Understanding the product will help you to become a better programmer. It's also a highway to higher positions in your company.

To get promoted, you need to learn what things does your company value.



If you could assign a $ value to the problems you're solving, be aware that the bigger positive value for the company, the more important the problem. If you have great ideas to solve a larger problem that others are working on, volunteer and try. If you make it, it's one of the fastest path to promotion.

Think about your employer as you'd think about the client if you were a freelancer. Clients usually aren't interested in implementation details. They want apps to do what they're designed for and to do it well. Minor improvements that took you days are irrelevant to them.

Your job is to provide solutions for your employer.

In your work, you should focus on the value you provide to the company. Are you working on essential features or some minor details? You may ask for a transfer to the team working on core business. You want to put yourself in a position where you can influence essential parts of the product.

Take your time to understand what products are essential to your company. Check what technologies are used for these products and try to learn them. Prove to your managers that you're able and willing to stretch yourself and help in critical projects.

“Developers who understand the business get promoted.”

Understanding the product will help you get a promotion and also grow your programming skills. Never forget that you're the problem solver, and you need to identify problems worth your time.

Another path to promotion is learning skills lacking in your company. Maybe your company rejects many projects because they don't have experts on some framework, data visualization, or anything else. Ask your boss what they feel would be useful. This way, you can show initiative and get guidance on which skills will be rewarded.

Coding is a part of the job. The higher your position, the less time you'll spend coding.

You need to understand the global scope of the product to be able to guide it through many seemingly similar paths towards success.

You can achieve only so much if you focus solely on coding. To be promoted to lead or architect position, you need to understand the product – its current state and direction. You need to plan and be able to prioritize.

Developers who understand business are valuable on every level, but they make the difference between success and failure in top positions.

Understand the business, and you will be the one who makes the difference.

Key takeaways

- ✓ your job is to provide value - the value is not your code, but its ability to solve a problem
- ✓ you should pick tools depending on the use-case, one size doesn’t fit all
- ✓ find out what’s the most important part of the product and focus on it
- ✓ find out what your employer values and master it - this way you’ll be promoted fast

11.

Developer in a Startup

Business
Side of
Startu
Busine

Developer in a Startup

UNDERSTANDING BUSINESS

All the work developers and designers are doing in a product is usually because of a business-related reason. There are of course those rare, artsy-installations where people code and design artwork. But if you want to use your skills to get paid, you need to understand how what you do impacts the business side of things. In this chapter I'll walk you through the basics of startups - how they operate, what stages they often have and how knowing all this can benefit you as a developer.

LET'S DIVE IN!

Joining a startup can be the wild ride of your life, or a difficult life-lesson. Either way - you win! Chances are, that joining a new company early on will lead you to be a part of the company, rather than just an employee. That's great if the company reaches unicorn status - it means it's worth a Billion dollars. But before you start picking yacht colors using the knowledge from the previous chapters, hear me out.

When working at a startup it's especially important for you to be versatile. Sure - doing your job the best way imaginable is the most essential, but knowing how what you do ties to everything else can make the entire experience a lot better.

The reasons are simple. For one, communication with the rest of the team will be a lot better when you know their **MVP's** from **ROI's** and **VC's**. But developers are primarily problem solvers, so knowing more about how startup works and how it makes money can actually lead to you contributing a lot more than just code.

Before finding creative ways to solve problems you need to know what they can be. Let's start with the vocabulary. These are the most common "startup" words you may hear.

MVP

Minimum Viable Product - It's the alpha version of the product that has the core functionalities, so you can test how users react to it. It's usually a way to test if the product even makes sense - or in other words: if people are going to buy it.

Bootstrapped

Contrary to some beliefs, this doesn't necessarily mean they used Bootstrap for the Frontend. It means that the product was done by the founders, without any external cash and they worked super hard on it. People wear it as a badge of honor if they manage to make it big.

VC Founded

VC stands for Venture Capital. It's basically a group of rich people who invest in small companies for a % share of them (called **equity**). Those VC's know that 9 out of 10 startups fail, but they hope on at least 1 of 10 being a huge hit. Being founded is a dream of many startups in the US, but it leads to less control over your own company and product. Those VC's will be your new bosses if your company takes them in.

ROI

Return on Investment - This is both a tactic for, and a time when a company is starting to actually make money (revenue). Many startups surf the wave of VC money and don't think about selling anything to actual customers, but avoid that route as it ends with a very sudden cliff while you're going really fast.

Acqui-hired
This means one company is buying another company. But their reason is to just get the people from that second company, so they likely start with handing out new tshirts and then killing the product they just purchased.

Incubator
It's a place that takes some % of the company in exchange for funding and/or teaching you a thing or two. This is called mentoring, but in many cases it's just sharing case-studies.

Burn Rate
If you get funded and cash starts flowing in, this is the rate at which you'll run out of it at your current monthly costs. So just divide the amount you got by how much you spend each month, and you will find out how many months will the company lasts. That is of course until it starts making money on its own, or gets another round of funding.

Exit strategy
It's your plan to monetize on the time and effort you put into the company. You may sell your shares, get acqui-hired or just go live off-grid in a van somewhere. It's up to you.

Angel Investor
This is a rich person that decides to give you money to pursue your dreams. They also take a share of the company, but are usually private individuals, not VC companies. They also often support companies very early on their ways.

Going Public
It means listing your company on an exchange, so individual (small) investors can buy them. Companies use it to try and get even richer, but it can backfire sometimes with very high expectations that are impossible to meet.

Pivot

A pivot is when a company doing one thing, decides they'll be doing another thing from now on. It can come from one feature of your app being used a lot more than the rest and becoming an app of its own. Pivots should be informed and backed by evidence, as doing them randomly can hurt the business rather than save it. And in many cases a pivot is being considered when the company is not doing that well. But in reality, when we look at the true unicorn companies, most of them worked on their initial vision of success. Only a small number of those "big ones" has been a result of a pivot.

NDA

Non-disclosure agreement. It's a form of agreement that companies and individuals sign making sure nobody is going to leak any secrets. It's also something you'll likely be facing in nearly any company that is going to hire you. It means you have to keep everything that's happening in the company to yourself.

Disrupt

It's what many startup founders crave. It means you came up with a product so innovative, it's changing (disrupting) the entire industry. Also: a TechCrunch startup conference name.

These are the most important terms to familiarize yourself with at the start. But don't worry - there will be more. It's important that you know the basic vocabulary, so no part of the startup journey is ever intimidating.

“

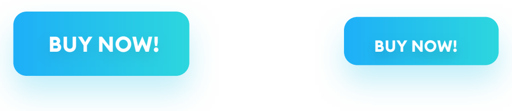
Joining a startup early means you should contribute more than just code to it.”

WHAT CAN YOU DO

Let's start with the basics. If you're in a startup and you own shares of the company, it should motivate you to do your best work. Don't expect the rest of the team to "make you rich in the background". I hope it's clear by now that you need to do your best work. And then some.

As a frontend developer you're responsible for one of the most important things in a product. It's all the stuff your users and customers **see and interact with**.

You can only make a first impression once, so make sure it's the best possible one. That means pixel-perfect precision, alignment and consistency of the UI. That super-high quality of execution is something your users will subconsciously **feel** and appreciate.


BUY NOW!
BUY NOW!

The buttons above are a good example. If the label on the button is not properly centered and the button is too small to comfortably click, it will create friction. The more friction the product has, the more it has to make up in value. And at some point there's no additional value it can add and your users begin to drop off.

Think about all the little details from the previous chapters. Make sure the grid is solid and every object is aligned to it perfectly. Use the right fonts, sizes and weights. Have a precise, consistent color palette. Avoid having random extra colors or gradients.

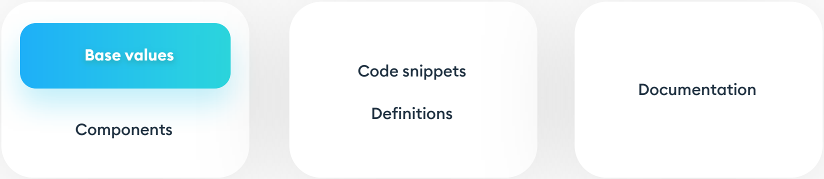
In other words:
Make your frontends look perfect!

Reusability and testing are another great way to bring your work to another level. Reusability is making sure you'll able to use your existing frontend components to build new things. The more you master these techniques, the faster you (and your startup) will be able to iterate and try new ideas.

That naturally lead to testing - both A/B and surveys are a great way of finding out what works and what can still be improved. Make sure to build your views with this in mind.

Design systems and standardized components allow for using the current design as the control group, with quickly generating new versions as the "challengers". If after some time the challenger does significantly better than the control version, you may have to switch to it.

Design systems can be intimidating at first, but you don't need to become the next Atlassian or IBM from the start. The goal is to organize your components to make them easy to find and as reusable as possible.



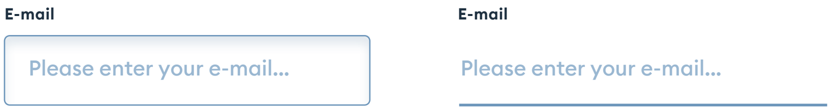
A typical design system starts with a design library. It's a way for designers to define all the base values (grid settings, font sizes) and create the styleguide for all the components. Then it's your turn to turn those components into code. The code snippets and definitions in the documentation are a way for others to pick up where you left off and help you build additional views and features. It's like a styleguide for developers.

SHOULD YOU?

It may be tempting - after watching the big, popular design systems - to go all in with your startup. But the reality is that a small, early-stage company doesn't need a design system just yet. Your goal, as a developer, would be to try and systemize the work, so it's more streamlined and consistent.

But it doesn't mean you should spend a year now putting a whole, big design system together. Just use "systemic" practices to work with designers and stakeholders. Document the components and keep the consistency as high as humanly posible. Of course there will be times when you have to deviate from the rules and that's ok too. Just make sure to be conscious of that and always have the consistency in mind. A product that uses the same, recognizable elements will be a lot easier to use and this will contribute to the products success.

INTERNAL CONSISTENCY



Internal consistency is a vital element of a successful product. It means the same kind of object within the product should always look and work the same way. One huge brand that we analyzed has completely different styles for their login and registration forms–all in the confines of one single app. That causes confusion and frustration in users. They were expecting something and got something different in return. If you don't push consistency from the start, it will only get more chaotic as the product grows. So be a guard of consistency at the company!

> “Make the design system fit your company instead of making the company fit an existing system.”

This is the most important advice I can give you. There was a stereotype of a developer being "in their own world" and only allowed to talk about development stuff. Don't fall into that trap! Yes - you are a developer, but you're also a human being with ideas and dreams. Coding is a great way to learn patience and dedication to solving complex problems. But you can also take that skill and use it on other types of problems. Don't be afraid to speak up and contribute in a small company. If they're talking about a new marketing plan and you think you have an idea don't assume it's "silly" because I'm **JUST** a developer.

In our startup adventures developers were often the ones coming with really innovative perspectives on tackling the problems well outside of their "regular scope".

Try to get out of the comfort zone of your scope as often as possible. Contribute!

Of course that doesn't mean you won't face criticism. Sometimes your ideas can be really bad, but that's ok. Get comfortable with feedback, but never get comfortable with being silent when you do have an idea.

Even a mediocre idea can actually spark another idea in some other team member and you'll be indirectly responsible for something awesome! The key to having good ideas, however, is to understand as many elements of the puzzle as you can. Focusing on just development skills may not be enough for that, so that's another reason why we made this book.

Knowing design, marketing, a little bit of business AND development will allow you to connect the dots and come up with awesome ideas!

Good luck!

Key takeaways

- ✓ most startups fail...
- ✓ ... but joining one that succeeds is the best way to grow as a developer - both financially and skill-wise
- ✓ whenever you can, try bootstrapping as much of the product as possible
- ✓ in a startup, you can't "just code". Learn as many skills next to yours as you can, because solving problems on all levels is critical to success.

12.

Master GIT

Master GIT

UNDERSTANDING GIT

This book is not focused on technologies, but rather on how to become a great developer. But still, we need to cover some basics, so you'll know how to work in any team, and make sure you avoid making a huge mistake.

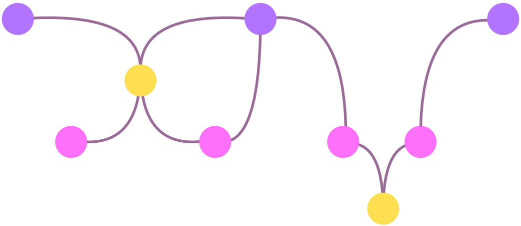
Version Control is the place where things can go terrible, so we'll teach you some basics, typical workflows, as well as techniques that can save you when you'll accidentaly do something awful.

Let's get started!

Code;
Code;
Code;
Code;
GIT

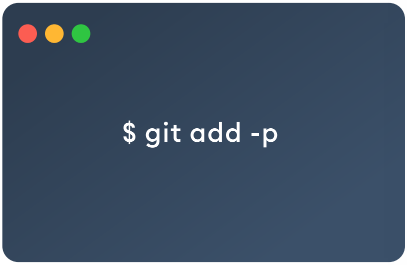
> “Using a version control system is non-negotiable. You just have to know it by heart.”

The crucial thing is that - you need to use a version control system. Always. It's non-negotiable. There are some alternatives to Git, but in reality, Git is the industry standard, so you'll have to master it.

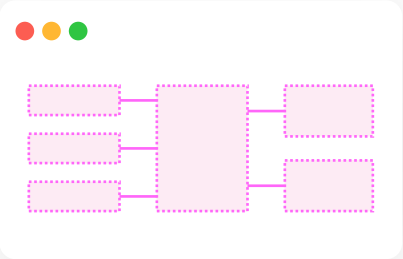


You can use Git by some GUI (a visual interface) or by the command line. Typically hardcore programmers advise using the command line, but in this case, I'd argue it's better to start with a GUI.

Some projects, especially with lots of branches, tend to get complicated, and having a visual representation can be useful. No need to overcomplicate things when not necessary.



```
$ git add -p
```



Nevertheless, you still need to know how to make some necessary actions in the command line, so we'll cover that too. It's always good to see the internals, and with a grasp of the command line, you'll be able to use the GUI of your choice even more effectively.

Don't worry about a steep learning curve. You only need a bunch of commands to start using Git in your project.

Let’s assume you created some code, and you’ve got a remote repository where you want to store the project. You’re in a command line at the root of your project. Here are steps to start with:

git init
the command creates a git repository locally. You need to do it before running any other commands.

git remote add origin /path/to/remote/repository
the command connects your local repository with the remote repository.

git add <filename>
the command prepares files to be committed to the repository. If you want to add all files, use git add *.

git commit -m "Commit message"
the command commits files to the git locally. The changes aren’t yet in the remote repository.

git push <localRemoteName> <branchname>
the command pushes changes from the local repository to the remote repository. You can use whatever localRemoteName you like; typically, we use origin. Branchame has to correspond with the branch name in the remote repository.

That’s all. Your local code is now connected with the remote repository, and the latest changes are pushed there.

To use git effectively you need a few extra commands, mainly:

git branch <branchname>
creates a new branch but stays on the currently active branch.

git checkout <branchname>
the command changes branch you're on. If you want to create a branch and change the active branch to id it, add -b flag (git checkout -b <branchname>). If you want to make the branch available to other remote repository users, you need to push it.

git pull
the command updates the active branch of your local repository with changes from the remote repository.

git merge <branchname>
merges another branch into your active branch.

git stash
the command reverts changes in the active branch but saves the changes you've made in the stash so you can get back to them later.
git stash pop - the command gets the latest stashed changes and applies them to the active branch.

If there is a repository online and you want to get it locally, you need to make sure you've got access to it by an SSH key or username/password and then run:

- **git clone /path/to/repository** - if your SSH key is added to the repository
- **git clone username@host:/path/to/repository** - if you need to log in using username and password

That's a lot to digest, I know. But on the bright side - most of the developers use nothing beyond it. Now let's see what some typical Git workflows in real projects are. Every company, or even every project, has its git workflow. It may be a conscious decision or just the sum of habits of team members, but there is always a workflow.

When you start working on a project you need to do is understand the Git workflow used So let's see how it may look.

Commits directly to master branch
It's the simplest way to work with Git. You may use it on personal projects. Unfortunately, it is also used in many professional projects. Its disadvantage is a lack of separation of work between developers, lack of a mechanism for code reviews, and too broad permissions for each developer.

If your company uses it, you should urge them to stop. Nevertheless, it's quite common practice, so you need to understand it.

The basic flow looks like this: You pull the repository and work with your changes. When you're ready to commit, you pull the repository again. If there are no conflicts, you commit your changes and push them to the repository. Done.

In case of conflicts after the pull, you stash your changes, pull the repository again, and pop the stash. Now you'll have to resolve conflicts by hand, commit code with resolved conflicts, and push it. And now you're done.
As you can see, it's a simple workflow. It's also error-prone. If you're not the only one working on the repository you'll constantly have to resolve conflicts between your code and code in the repository. And you may not have sufficient knowledge to do it, or you may accidentally break some things.

> “Unless your company enforces it, avoid committing directly to the master branch.”

To resolve those problems, many teams use more complex workflows.

Feature branch

With a feature branch workflow, every feature gets a new branch. A developer working on the feature commits changes to that branch until the feature is done. Usually, he's the only one using the branch, though, in bigger teams, more programmers can work on this same feature.

When the feature is ready, a developer merges the branch with the master branch.

This workflow is an upgrade on committing directly to master because the only moment when there are conflicts between branches is when two of them are merged. Other than that is has these same disadvantages as committing directly to master.

Feature branch and pull request

It's a similar workflow to the feature branch, but it gives more granular control over each developer's permissions. Now every feature has its own branch, but not every developer can merge the branch with master.

To merge the branch with the master, the developer has to issue a pull request. The request is verified by a senior developer who can review code, add comments, and return the branch to further work or approve it and merge.

The advantage of this approach is the ability to give different permissions to different developers. This usually leads to more testing and better quality of the code overall.

Git flow

Git flow is one of the more complex workflows; it's usually used for applications already in production but still developed.

In this workflow, we've got at least two separate branches being used all the time. One is for the production application, and it's usually called master. The second one is used for development, and it's called develop.

The develop branch is used similarly to the master branch in a feature branch and pull request workflow. Developers work on code on their branches, and the code is then reviewed and merged with develop by senior programmers.

When the new version of the application is deployed, the develop branch is merged with the master branch.

The described approach is excellent for developing new features, but what about bugs you need to fix in the production application quickly? For that, Git flow offers hotfixes. To make a hotfix, you make a branch from the master branch (not develop branch, like new features).

When the fix is ready, you merge it first with the master branch, and only after that, you merge the master branch with develop.

Git flow is a popular approach, so there is even a tool that can help you automate creating and merging branches. You can find instructions on how to install it here. The link will also be available in the resources section at the end of this book.

Now and then, you'll break everything on git. Your merge will be a disaster; you'll commit many changes on the wrong branch or delete crucial files. Is there a way out? Sure, that's one of the reasons to use version control.

You've accidentally deleted a branch or a file
When you've deleted something, it can be hard to get it back. In that case, you can use git reflog to get back in time. The command will show you every local change you made to all branches with its index. You can now choose where you want to get back in time using git reset HEAD@{index}.

You've committed files to the wrong branch
To fix it, you need to remove the last commit from the active branch and keep the file changes in the working tree. You can do it with git reset HEAD~ --soft. Now you just need to go to the correct branch, pop changes out of the stash, and commit them. You should be able to do it yourself by now, but just to make sure:

```
git stash
git checkout branchname
git stash pop
git add .
git commit -m "Commit message";
```

You want to revert a commit
That's easy. You just have to find the commit hash with the git log. Use git revert <hash> to create a new commit that will undo the one you want to revert. Be wary this may generate merge conflicts, so it's better to use it only for fresh commits, not something from two weeks ago.

You want to see how to code looked like at the moment of a specific commit?
Just find the hash of the commit with git log (or in the GUI). Then use git checkout <hash>. Done.

When talking about Git, we can't forget about .gitignore. This is a file, where you specify the files and directories you want to exclude from the remote repository.

Code;
Code + Junk

What do you need to exclude? Well, the list is long and varies from project to project, but here are some ground rules:

.env files and other vulnerable keys, data, and credentials - that's the most important. You don't want to have private information in your repository - it's a huge security risk.

node_modules and other files downloaded by package managers - they take a lot of space and should be installed separately by every developer.

operating system files - file like .DS_Store are not important for the project; your operating system adds them.

cache - can be huge and is not needed for other developers.

build output directories - you should build production versions of the application on the server.

There may be other files ignored in your specific project. The general rule is to include only files necessary for the project to run (not created on the build), and that are not vulnerable.

> “The first rule of GIT is: MAKE MANY COMMITS.”

Mastering git is hard, and you will make mistakes. Senior developers in your team should be aware of that, so it's not a huge problem. But you want to focus on making their job easier and don't make them mad needlessly.

So I've got one final advice for you.

Make many commits.

The smaller the commit, the easier it is to test and review. No one will want to check your week's work, where you changed dozens of files. Try to stick to changing at most a few files and make commits at least once a day. Your team will be grateful.

Key takeaways

- ✓ you definitely have to learn and use Git in your work
- ✓ when joining a project always learn what is the Git workflow of your team
- ✓ Git can save you when you made a mistake
- ✓ make many small commits, the project history should be fairly easily read from commit messages

13.

Work-life balance

Work-life balance

WORK HARD, BUT NOT TOO HARD

You have to work hard to become a developer. Even harder to grow, change jobs, or get promoted. But to work hard in the long term, you have to take time to rest and decompress.

Having a balance is essential to being a great developer. Your personal and professional life has different priorities, and you need to find a good combination of both of them.

You're in it for the long term. Maintaining a healthy relationship with your work is not a choice; it's a necessity to avoid burnout, depression and more.

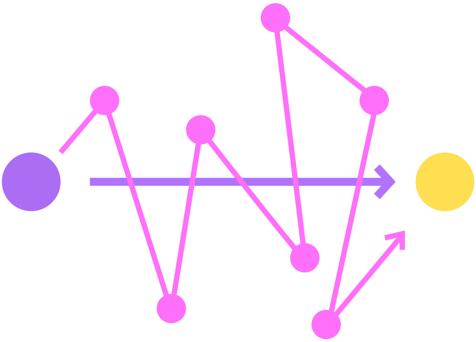
Outside of work
At work

Some people can code all the time; they love it and don't burn out. But most of us are not like that; we need that real-life™ to be fulfilled and to grow.

And also, the better you are at what you do, the more efficient you can be. That means less repetitive work and the balance being able to shift more towards the "life" spectrum.

Work, learning, personal life. At different points of your career and life, you'll have other priorities, needs, and capabilities. Your job is to prioritize these three areas of your life so you achieve the most.

You have to decide where you are now and what your priorities should be at the moment. But let me help you to approach your time more mindfully.



You can either go straight towards your goal, or choose a longer path full of distractions and things that don't contribute much to the experience. Your choice!

This chapter has one central concept - quality over quantity. No matter how you want to use your time at the current stage of your life, make that time count. Work hard, try automating, and delegating everything not essential. Learn smart, don't rewatch these same tutorials again and again.

And when you rest, do it properly, don't just scroll Facebook or watch Netflix; give yourself a chance to rejuvenate.

> “Try automating or delegating all the non-essentials.”

You have to know your limits. Most of us can't work 60 or 80 hours a week for prolonged periods.

I know there are times when you don't have a choice. You've got the release of a significant project, and your team counts on you. Your manager asks you to finish some critical features on the weekend. The crucial website crashed in the middle of the night. These are valid reasons to work overtime.

Working overtime before the release, is so popular it has its name - crunching. It's hard to decline when you're asked to stay overtime because the product's or whole company success may depend on the project.

It's okay if crunching happens from time to time - sometimes life is hard, and you have to grit your teeth. But if crunching becomes the norm, you've to talk with your manager or even consider changing a job.

In reality, working overtime is rarely beneficial. It's detrimental to your ability to focus and do your best, and it also negatively affects your company. Overworked developers ship worse code. Worse code leads to even more work with bug fixing. And quick bug fixing leads to even worse code. That's a vicious circle and good companies need to break it.

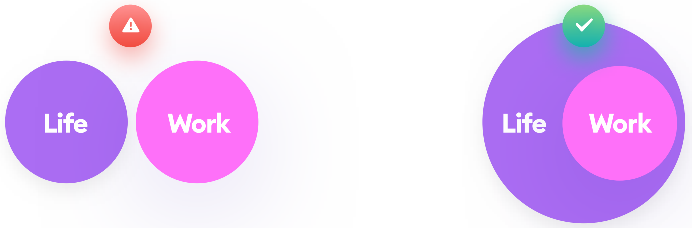
The benefits of putting more hours decrease over time. Whenever possible you should work only up to 40 hours. And work hard - be focused, don't overuse social media. Do your job to the best of your abilities.

If you give your employer honest 30-40 hours a week, you'll be one of the best developers around. Expecting more of you is unreasonable and unrealistic. A healthy and rested programmer is the best asset for any company.

So you work eight hours a day and can't wait to get out of the office? And the first thing you think about on Monday is a Friday evening when you can get out and live your life? Are you really going to do that for the next few decades? I don't think it's a great way to live your life.

Many people don't have a choice. They live in poor countries or are stuck in simple jobs without a real opportunity to live their lives differently. But you as a developer can do so much more. You can change your career, build your app, start a company, or become a freelancer. So don't allow yourself to live a miserable life from weekend to weekend.

The divide between a job and a life is tempting. You work to earn money for your life. But the truth is, that work takes a considerable part of your life. They're not separate - one is a part of the other. You can and should make it more enjoyable. Stop treating work as a struggle; look for opportunities to make that time count and to make it fun.



Maybe you can ask your manager to work on more challenging projects? Or you'd instead work some time remotely to avoid the commute. You can make your working environment more pleasant, but you need to ask.

Your main goal is to change the way you think about work. Try making it not the "bad part" of your life but the essential part of it. If it means changing a team or a job, think about doing it. In the long run, work that's not meaningful to you will become exhausting. Look for new challenges and opportunities to help others. Make your work count.

You matter. Forget your projects, colleagues, commitments. These are essential, but you can fulfill your obligations only if you take proper care of yourself. That's why you need to make yourself a priority.

It may sound harsh and selfish - you may feel like a bad person. But it's not. You have to be on your best and grow to provide value for those around you.

Social life
Learning
Health
Work
Exploring
Regeneration

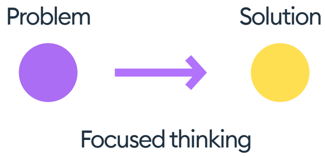
So make time for you. Work out, read, sleep, and learn. Being in good physical and mental health is your greatest asset. All of these are important!

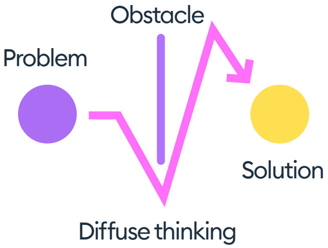
Many times you'll be tempted to stay at work late. To learn a new language for a few hours after a hard day at work. Hold back. Don't let your ambition get in the way of fulfilling your long-term goals. Tomorrow will be the next day when you need to be the best version of yourself. If you pull an unnecessary all-nighter, all you'll achieve is a bad next day. Or even a week. So prioritize well-being and promising effects in the long run over short-lived success.

You're in this for life. Prepare for a marathon, not a sprint.

Did I convince you? Do you believe a massive part of your success as a developer is a rest? If not, think about the last time you've solved a difficult problem. Often you spend hours or even days trying to crack the issue, just to discover that's impossible. And when you're ready to give up and go for a walk or read a book, the solution suddenly pops to your mind.

Why is that? It's because we have two modes of thinking - focused and diffuse. When you are trying hard to solve a problem, you're in the focused mode. That's good. It's the logical mode, used for analytical thinking. But it has some limitations - it's only great for the issue at hand. And some problems require creativity, not logic only. And for those, we need diffuse thinking.





Diffuse thinking is less structured, more focused on the big picture, and different concepts. It lets you create unexpected connections between subjects. Most of the diffuse reflection happens when you stop doing real work. Because in reality, your mind is still partly occupied with the problem you were analyzing. That's why great ideas come to us when we're taking a shower or jogging. In other words, creativity happens when you're resting.

To work to the best of your ability, you need to switch between focused and diffuse thinking modes. The best way to achieve it is to work hard, with a full focus. And then let go, rest, and have some fun. That will make your life more complete and your work better.

> “Try to use both the focused and diffuse modes of thinking to tackle difficult problems.”

Key takeaways

- ✓ it's okay to change your priorities during the career
- ✓ you can't work all the time, resting is an essential part of being a great developer
- ✓ focus on work quality, not on longer hours
- ✓ the best way to solve a hard problem is to tackle it with focus and then take a step back
- ✓ your time is important, protect it

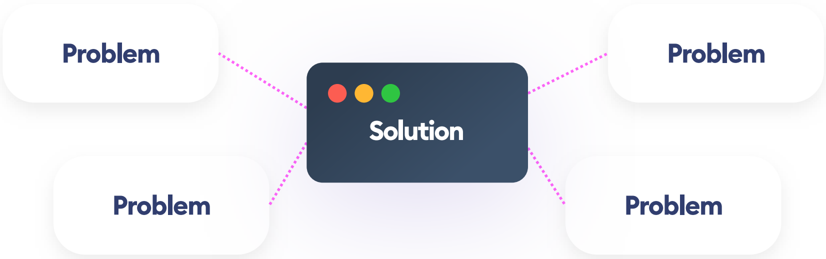
14.

Taking notes

Taking notes

A NOTE TO REMEMBER

I'm sure you're not taking enough notes. We all overestimate our ability to remember things. And fail again and again at similar tasks. You'll have to solve the same or very similar problems all the time. If you don't save your solutions within your reach, you'll have to reinvent them every time.



Many problems can have the same or a very similar solution.

Don't trust your memory and document everything—the ways you solved problems, useful algorithms, and functions, suitable learning materials. Pay attention to websites and people that helped you solve a problem.

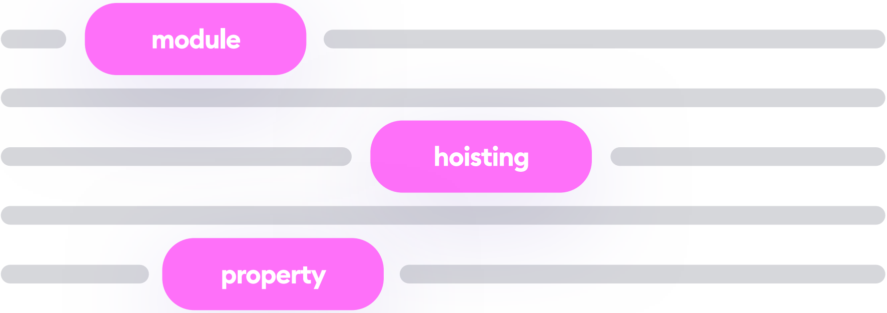
We are all different and learn in various ways. Some resources may be useful for you and useless for others. By saving everything valuable you'll build a robust database of reliable resources. Treat it as a personal library that will serve you for years.

It's easy to get lost in the sea of hyperlinks and other resources. Take some time to pick the optimal way to keep your notes. I use the app that allows you to add tags to resources and find them easily, but stick to whatever works for you.

> “You often encounter the same or similar problems all over again. If you don't document your solutions it may result in a lot of extra work.”

REMEMBER TECHNICAL TERMS

I used to think that I don’t need to know how something is called to understand it. After all, you don’t need to know the name of the phenomenon if you know how it works and can predict its behavior. For example, I know how gravity works, even if I don’t know it’s called gravity. But in reality, knowledge of technical terms proves a lot more beneficial than you think. Programming is a cooperative process, and programmers have a particular language to communicate fast and accurately. To find the information, you need to understand that language.



For example, it’s a lot easier to discover why you don’t have access to the variable if you know the term “scope”. Without that knowledge, you can’t ask the right questions. Besides, when you know proper terms, you don’t need to read so much. Skim the internet. Search for the keywords in the text instead of reading everything.

You’ll communicate a lot with other programmers. To find a common ground, you need to use appropriate language. It will also prove that you’re “one of us,”. A certain kind of pride in the precise language is common amongst programmers.

Note every technical name and define it, if possible, using your own words.

NOTE-TAKING METHODS

If you don’t have a habit of taking notes, it can be hard for you to start. Experiment with different note-taking methods to find the one that most suits you. Let's go over a couple of the most popular ones together.

THE OUTLINE METHOD

It’s the most straightforward method. You just have to organize your information hierarchically using points, bullets, letters, etc. For example:

OUTLINE METHOD - NOTE-TAKING	
PROS	**CONS**
• Easy to do • Focused on relations between pieces of information	• Not useful for maths, chemistry, or graphical information • You need to know the relations between information when you start.

THE CORNELL METHOD

This method makes use of tables. You need to split the page into a cue column, note-taking column, and summary column. You can also add a row for the title.

Cue column is narrower, and you put keywords and questions there. The Note-taking column is for notes. The title and summary are self-explanatory.

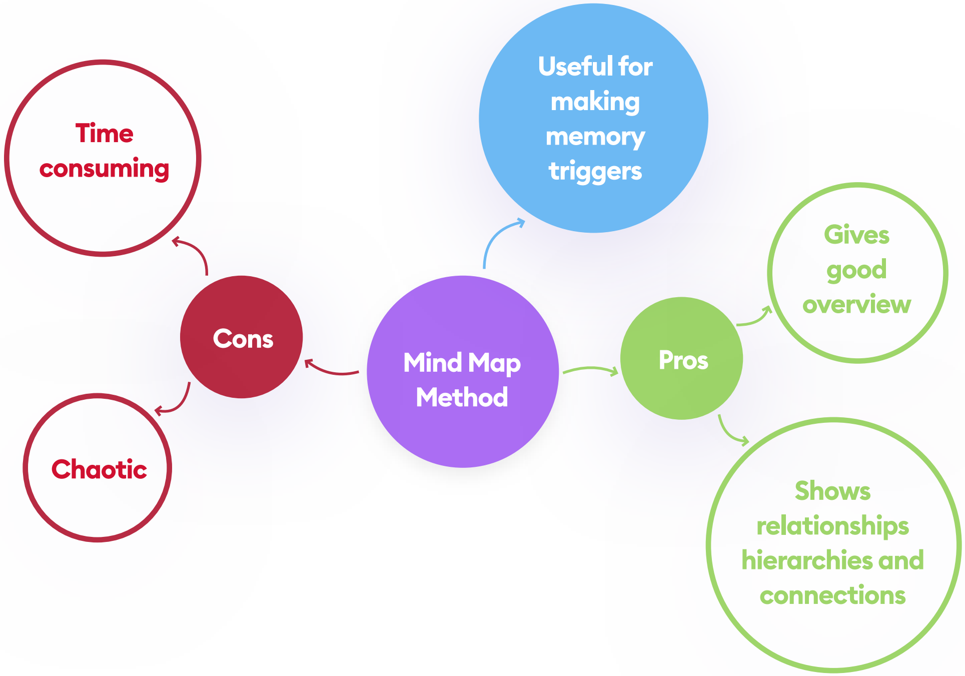
Here's how it can be organized:

THE CORNELL METHOD	
Keywords	notes, productivity, learnings
Pros	• Makes it easy to focus on main ideas • Good organizing tool • Flexible
Cons	• Takes more time than an outline method • Requires discipline to get started
Summary	Good note-taking method for a wide range of use-cases.

This method requires you to create a single “umbrella term” in the middle of the page. Draw arrows from the terms with your associations. Expand those associations further until you’ve got a complete map of thoughts about the subject.

Use different colors and lines to emphasize important information. You can place the information in boxes, circles or triangles depending on the type. In our example subcategories are made with outlines, while the main categories are filled.

You can draw mind maps by hand or use one of the online tools. There are some tools specifically created for mind-mapping, but you can use any tool that you're comfortable with, including PowerPoint, Keynote, Sketch, or Figma.


Time consuming
Useful for making memory triggers
Gives good overview
Cons
Mind Map Method
Pros
Chaotic
Shows relationships hierarchies and connections

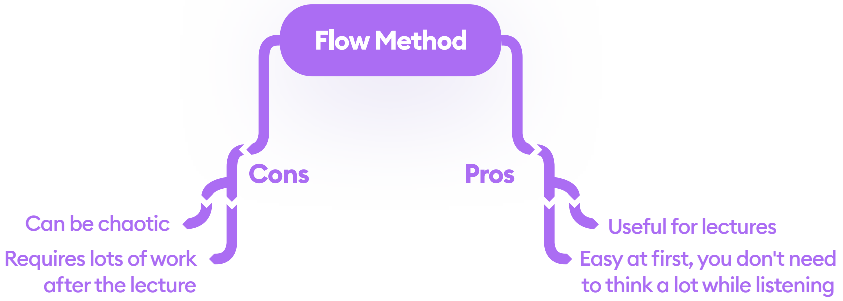
“You can use any software you already know to create mind-maps. Powerpoint, Keynote, or even design tools can also be used to organize information.”

Using the flow technique, you don't organize information at first but write essential details as they come up. Then, you make arrows and links between information to show connections. You end the process by synthesizing the information in your own words (that's important, no parroting). Optionally, when you get more information, you can backlink it to the existing notes.

Example notes from a lecture:

- Note-taking is important.
- The flow Method is one of the exciting methods.
- You make notes as the lecture goes.
- May be chaotic
- Useful for lectures
- Requires lots of work after a lecture
- Easy at first, you don't need to think a lot when listening.

You can organize the information above like this:

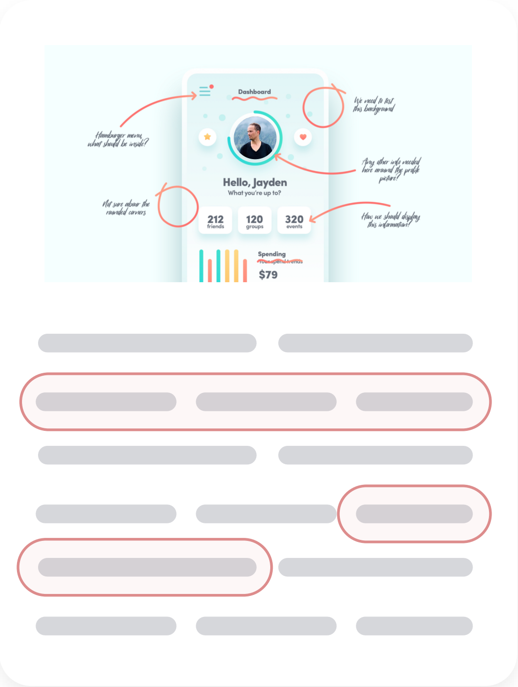

Flow Method
Cons
Pros
Can be chaotic
Requires lots of work after the lecture
Useful for lectures
Easy at first, you don't need to think a lot while listening

And summarize it like that:

A flow note-taking method is an excellent tool for learning. It requires a lot of time but allows you to make notes on the fly and then organize them into comprehensive data structures.

THE “WRITE ON SLIDES” METHOD

This method is pretty simple. You just take slides or notes from the lecture or article and write notes on them. It's a popular way to learn at universities, but can also work online with the various annotation tools at our disposal.


Dashboard
Hello, Jayden
What you're up to?
212 friends
120 groups
320 events
Spending
$79
Hamburger menu, what should be inside?
We need to test this background
Any other info needed here around the profile picture?
Not sure abour the rounded corners
How we should display this information?

You can highlight parts of the text, using color codes for types of information, draw shapes on illustrations and more. It's good, however that you come up with a consistent way of taking these notes, as it's going to make it a lot easier when working with multiple annotated materials.

Pick a style that works best for you and stick with it.

> “Experiment with all the note-taking techniques and find the ones that work best for you.”

Key takeaways

- ✓ technical terms matter, write them and try to remember
- ✓ taking notes is crucial, otherwise, you'll find yourself searching for solutions for these same problems again and again
- ✓ there are many note-taking techniques, pick the one that suits you best, but first try them all, as some may be better for specific use-cases.

15.

Time management

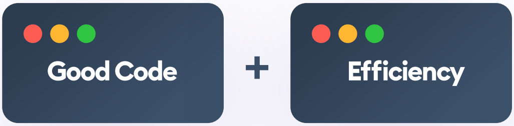
Time management

HOW TO WORK WITH DEADLINES

There is a saying that programming is all about trade-offs. You have to take multiple aspects of your code into account. It starts when you choose a language to learn - do you prefer a more robust and "to the metal" language, or maybe you prefer a more abstract and simple but less powerful one.

And it's just a start. You often need to decide between simplicity and performance, execution speed and ease of use, and so on.

Many programming courses cover those technical trade-offs, and you can quite quickly get a general idea about how to balance the priorities. But the programmer's training often doesn't consider the most critical asset you have and need to optimize for - time.



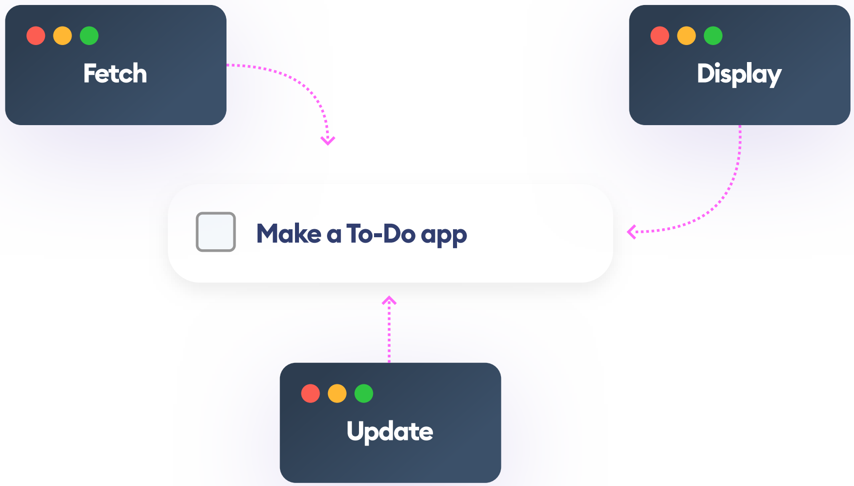
Time management is one of the crucial skills in any field. Just think about it. Imagine you are a part of the team building the next great Todo app. Your next task is to implement a component responsible for fetching, displaying, and changing a task. Your deadline is in two weeks.

This comes with countless time-related decisions ranging from scheduling, optimization, and maintainability to the decision on how long to work on a tricky part of the code and when to ask someone for help. Let's get through all of that process one bit at a time.

“Time is the most critical asset you need to think about.”

To be a productive developer, you have to know what exactly needs to be done. It doesn't matter how good your code is if you misunderstood the requirements. Being productive is different from being busy and hardworking. You are productive if you're doing the things that push the project further.

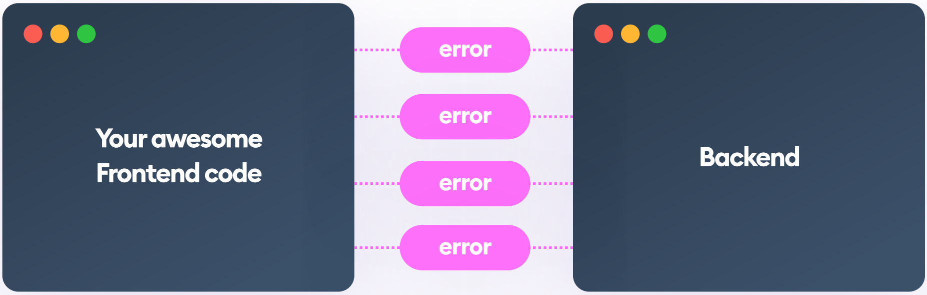
For your task in the Todo app identifying essential features is easy. You need to fetch, display, and update data. But there are more unknowns here than you think.



Ask what tools should you use for making API calls. You may use your favorite stack to see the rest of the team uses something else and need to rewrite the code. Make sure if you're are responsible for validating or escaping data. Find out if you need to implement the design with pixel-perfect precision or do you need only a more or less accurate interface. Get as precise information as possible, so you avoid wasting time.

You should schedule your work with a significant rule in mind - you want to know about the possible major problems first. Things you cannot control create the greatest problems. That often means you need to look at things that depend on other people and their work.

The first thing you need to do is to analyze the API endpoints. Ensure you know about all the necessary endpoints, check if they work, and return data or errors as expected.



The above example may be a bit exaggerated, but trust me - more often than not this is exactly what happens.

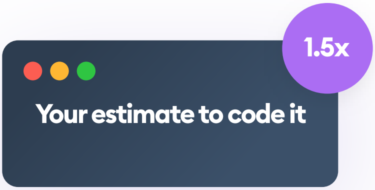
More often than not, something won't work the way you need it to. You may need help from the backend developers, and you don't know if they have time to do your bidding.

It may take a few hours or days to fix a backend issue, and waiting may block your progress, so it's crucial to start with it. That way you can figure out a workaround, or work on other areas of the product, while waiting for the backend guys to address the blocker issues.

The second priority are complex algorithmic and structural problems. Think about all the algorithms and data structures you'll need to use in the project. In front-end development, challenging algorithmic problems are relatively rare, but make sure that you don't need to traverse trees or make a search functionality.

Complex data structures are more common. Usually, the data you get needs some transformations; you may need to use different structures for UI and requests.

Most of the time, you should leave implementing the UI for the end. HTML and CSS are usually the simplest. Also, from the manager's perspective, they're not as crucial as the core functionality.


1.5x
Your estimate to code it

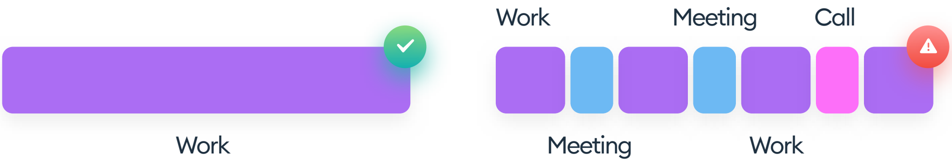
I've got two final pieces of advice on scheduling. Everything takes more time than you expect. So when planning your work, assume coding will take 1,5 times more than you think. Also, don't forget you're responsible for testing your code. It doesn't matter if your company has testers; you want to ship software that is tested and works fine. Testers are there for edge cases, not basic lookup.

After examining all aspects of your task, you should have a general idea if it is achievable with your skillset and within a given deadline. If you believe you can't do it, let your boss know about it as fast as possible. Don't be afraid; most managers can handle respectful and sensible reservations. What they can't handle is a lack of reliable information on time.

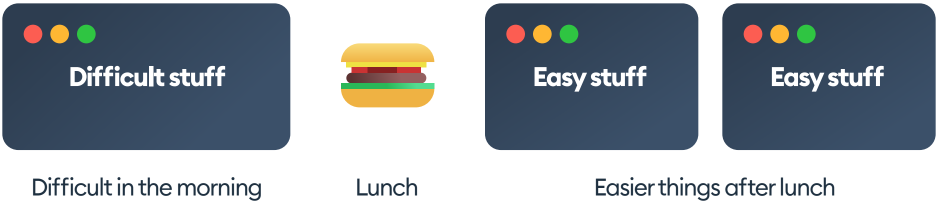
> “Estimate the time to code something. Then multiply it by at least 1.5 ”

Okay, so you've got a general idea of when to work on what. But how should you approach each day to maximize your full potential?

Start your day with the most demanding task. You're likely to work the best when you're well-rested, and your mind is fresh. Don't wait for inspiration, don't answer emails, avoid procrastination at all costs. Get to the problematic code as fast as possible.


Work
Work
Meeting
Call
Meeting
Work

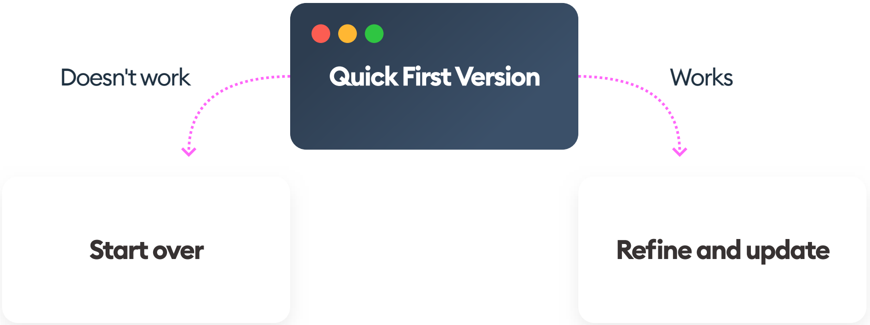
For difficult tasks, try to schedule longer periods of uninterrupted time. You can achieve much more in 90 minutes than in six 15-minutes long bursts interrupted by calls, emails, and meetings.


Difficult stuff
Easy stuff
Easy stuff
Difficult in the morning
Lunch
Easier things after lunch

After lunch, in the second part of the day, it's better to do less demanding, more repetitive tasks. It's a great time to make calls and answer emails. After coding in the morning, you probably have some code to test, so do it. Inevitably there will be moments when you get stuck. You'll think about how to solve the problem and feel every solution seems terrible. You have to remember that at first, you don't need to make your code perfect. It's enough to create any code resembling the final solution. Done is better than perfect – at least for now.

Write your first draft as fast as possible. Maybe it won't be any good, but you'll explore some options and check if your approach is viable at all. Don't be afraid of failure. Assume you'll fail at first. Fast feedback is the best feedback. If your method failed, try another one. If the code works, you can build on it.

Also, remember about the priorities. Don't get hung up on small details. If you can't build a UI element or can't implement a minor feature - leave it be. You'll get back to it when you're fresh. In the worst-case scenario, inform your manager that for now, it can't be done.



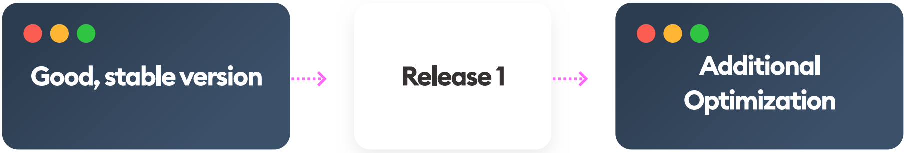
In many cases, you'll be waiting for someone. Maybe it's a backend developer who needs to create an endpoint or a designer who forgot to prepare icons for you. Make the most of the waiting times. Don't take a break; move on to different parts of the task, or test your code. Developers work under time pressure, so don't waste time for no reason.

It doesn't matter how good your work is; sometimes, things go off. If you know you won't finish on time, let your manager know as fast as possible. Missing the deadline is bad, but it's much worse to know you'll miss the deadline just half an hour before hitting it. Deadlines are important, but with good communication, they're negotiable.

The secret of efficient coding is the focus. Implement only required features, make as few assumptions as necessary.

It's common for developers to create solutions "just in case." They assume the next feature may be needed and prepare their code to be developed further. It prolongs development time and adds complexity. And then the project shifts unexpectedly, and all additional code becomes unnecessary and even harmful.

You should always have performance in mind and use good practices. But you have to avoid optimizing code in the early stages of the project. Developers spend countless hours improving the performance of performant enough code. All code can always be additionally optimized, but most of the code shouldn't.



Only when the project is mature, it's time to maximize performance. This way, you can find significant bottlenecks and start optimization with them. Few millisecond improvements in other places are rarely worth dozens of development hours.

Use a similar rule of thumb when deciding how good your code should be. We all strive for modular, reusable, and maintainable code. But you can't write the perfect code; some things can always be improved. And time for the project is always limited. So try to write as good code as possible and always leave time for refactoring. But don't get stuck in endless refactors. Enough is enough.

The project you work on is often vital for the company, but your own progress is crucial for you. Regularly you’ll encounter problems you don’t know how to solve right off the bat. If you struggle for an hour or two, you have a choice - should you try working longer or maybe ask someone for help.

You’re much more likely to learn new things if you’ll try to handle problems by yourself. But you can’t impede the project by spending countless hours on simple features. Usually, the right balance is to ask for help after a few hours. If you’re stuck for two or three hours, leave the task and get back to it the next day. If after another two hours you don’t have the solution ask others.

This way, you can reap the benefits of struggling with the challenge without slowing down the project. Of course, if your team is working on tight deadlines, the project is the priority, and you’ll need to ask for help faster. But as a general rule, remember that projects come and go, but your knowledge stays with you forever.

Key takeaways

- ✓ time management is all about setting your priorities right
- ✓ start your work with the tasks that depend on others
- ✓ if you know you won't deliver on time admit it as quickly as possible
- ✓ get your hands dirty fast - no action plan survives first contact with the enemy (a difficult problem to solve)

16.

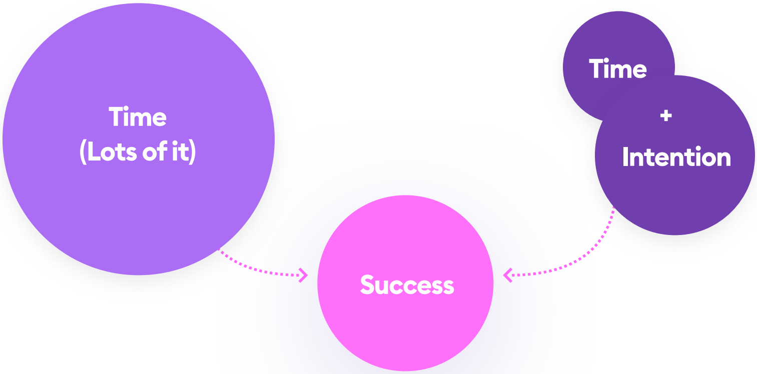
Code review

Code review

CODE REVIEW IS YOUR FRIEND

What’s the fastest way to learn? Practice will bootstrap your career in almost every field.

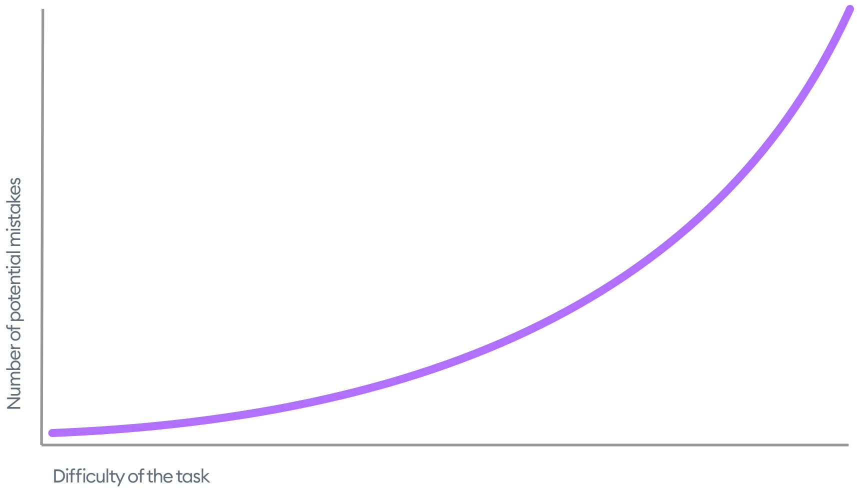
Malcolm Gladwell famously stated that you could master any skill in 10,000 hours. While this can be true in some fields, it's definitely not universal. Time alone is not enough. That's because not every practice session is made equal. You can smash a ball repeatedly at the goal but never become a decent soccer player. What matters is deliberate practice.



You can often get the same or better results with less time.

How can you make your practice deliberate? Deliberate practice is systematic and purposeful. You need to code daily to become a great coder. And you need to think about what you’re doing, continuously increasing the difficulty of tasks ahead of you. Coding mindlessly the twentieth landing page won’t let you improve; you need to stretch yourself to progress.

Deliberate practice is all about solving increasingly more complex problems. But with more challenging tasks, you're bound to commit more mistakes. You need to identify and fix those mistakes. It's often hard, as you don't know what you are doing wrong. You frequently don't even know you're making a mistake at all. Fortunately, other people can help you.



You can't take full advantage of deliberate practice by yourself. You'll need guidance from others, preferably more experienced in the field. Your mentors and teachers are crucial for progress as a developer.

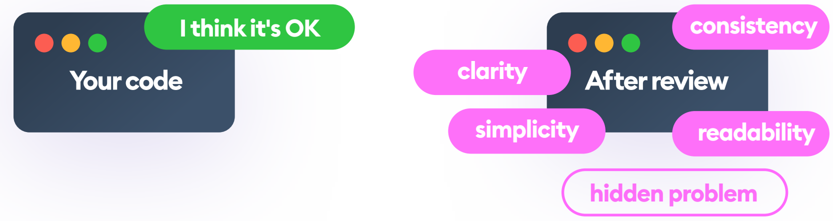
Nobody is a better teacher than a senior developer working on your team. Seniors can give you unique feedback because of their knowledge about problems and pitfalls you couldn't predict. In simpler terms: they often have seen it all - including some scenarios so bizarre, you wouldn't be able to predict them happening.

The most convenient way to get feedback from senior developers is by code review.

Code review is a software quality assurance activity in which one programmer checks other programmer's code. To put it simply, usually, you write some code to implement a feature and test it manually. Then it goes through automated tests. Before merging it with the master branch, the code can be reviewed by a senior developer.

Code reviews are a good practice, and every software company should have them in one way or the other. They help to build better software and avoid silly mistakes. But what's vital for you, they provide invaluable feedback for junior developers. Even if your code is working correctly, you may learn a lot in only 20 or 30 minutes of conversation about it with a senior developer.

Another pair of eyes can help you to improve your code a lot. Making the code work is just a small part of the task. There is always much space to improve readability, simplicity, and consistency. You also have to account for performance and exception handling.



Code reviews can be painful. It's no fun to listen about your mistakes. This makes your mindset crucial. Every code review is an opportunity to learn. Every mistake is a pathway to progress. It's hard to enjoy criticism, but at least try to be open-minded about it.

> **“Treat every code review as an opportunity to learn and grow.”**

The sad reality is not every company performs code reviews. It's not unusual for junior developers to push the code directly to the master branch. Often you're the only person who has seen your code.

It doesn't mean you work in a lousy company. There are lots of reasons why a company may skip code reviews. Especially in front-end development, programmers produce a lot of code. Thorough code reviews are very time consuming, and it's hard to analyze longer pieces of code properly.

Additionally, senior developers often have many other duties besides coding. They may lack mental capacity for different tasks. Also, juniors are to blame, as they frequently impede reviewing code.

Hey, can you look at my code?
My code

So, how can you get feedback when your company doesn't have regular code reviews? Ask! Asking for reviews is one of the most essential traits of a junior developer who wants to be successful. It's easy to ignore you, but it's much harder to deny your code review request.

Ask your manager who can review your code and then set a date and scope of the review. You need to be assertive and don't allow your code reviews to be rescheduled all the time. They're essential not only for you but also for your employer.

If your company doesn't have a code review practice and your seniors are reluctant to help you, we've got some tips to make things a little easier. You should help other developers to review your code. To achieve it, you need to understand how a typical code review looks like.

Code review usually takes between twenty and sixty minutes. It should be quite thorough, so it's impossible to go through many lines of code or multiple files. Also, if you've spent on a code two weeks, it's improbable that someone will be able to analyze it in less than an hour.

The reviewer most often will compare code from pull request with the code in the master branch. So it's crucial for you to have the least amount of commits in the pull request (preferably one). This will make the lines of code you've changed or added clearly visible.

Also, the developer probably will stop review if your code is not working correctly. It's your task to check if the code is ready to review and possible to understand without Googling anything.

In short, the senior developer won't have much time, so you make it as easy for them as possible. Here are some tips to help both you and your reviewer.

Firstly, think about what parts of the code should be reviewed. **You don't want to use your seniors' precious time to check simple CSS or other trivial code.**

In a typical application, the most challenging part of the front-end concerns state management, interconnections between components, and data fetching. Feedback about these tasks presents you with the most significant chance for improvement.

> “You need to ask for feedback yourself. Do it often.”

Also, don't bite more than you can chew - depending on the speed of your programming the code to be reviewed should take from a few hours to two days to create. Preferably code shouldn't be longer than several hundred lines of code and concern more than 3-5 files.

Now you know what code needs a review—time to make it review-friendly. Try making the code as readable as you can. Add comments in harder to grasp parts of the codebase. If you're concerned about a particular piece of the code add information about that.

If possible, place all of the code you need in one commit. Necessarily use the same code formatting rules as your team. Otherwise, even adding one space in a file can result in changing everything, as your code formatting tool will rewrite the whole file.

```
-    <Col>
-      <div className="customSearch">
-        <SearchBar {...props.searchProps)>
-          <ClearSearchButton
-              className="customClear"
-              text="x"
-              {...props.searchProps}
-            />
-       </div>
-    </Col>
-    <Col>
-      <div className="flexEndContainer">
-         <ExportCSVButton {...props.csvProps}>
-            Export CSV
-         </ExportCSVButton>
-      </div>
-    </Col>
```

It was just a typo in the file, but a developer used four spaces instead of two, so the whole file was overwritten.

If you prepared everything for the review your job is done. You've made things as easy for the reviewer as possible. It's time to receive feedback. Don't be afraid of criticism; use every bit of new knowledge to grow as a developer.

Key takeaways

- ✓ code reviews are stressful but they're also great opportunities for growth
- ✓ if your company doesn't provide code reviews you should ask for them
- ✓ prepare your code for the review to make the work of senior developers easier

17.

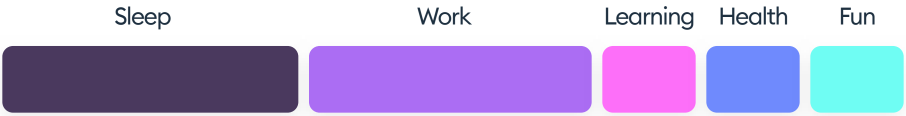
How to learn effectively

How to learn effectively

LEARNING TO LEARN

Time is the most important asset you have.

Imagine having all days to develop your skills. Someone could take care of such mundane things as cooking, going to the gym for you, earning money and paying all your bills. Becoming a good programmer in such circumstances would be a piece of cake.



But the reality is different. Very few people can devote their entire days to coding. Do you know anyone who has two or three hours to spare daily? Yeah, me neither. That’s why you have to make good use of your learning time. The more effectively you can learn, the less time you need to spend learning. And the greater is your chance to succeed.

Learning to learn is the most critical skill to acquire. In this chapter I'll cover the key elements to learning programming faster. That way you can maximize the limited time you have for learning each day. You can use the same amount of time, to simply learn significantly more.

Learning time
Learning outcome
Learning time
Learning outcome

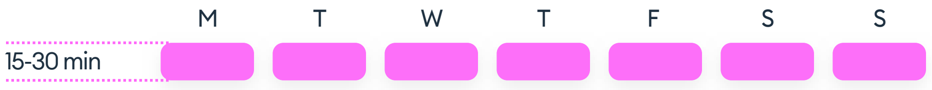
You work hard for a week, but now it's Saturday, and you've got four hours just for you. An ideal time to do some coding. So you open a video course that you bought and you start learning. Some hours pass. You're exhausted, but it was worth it; you've learned a lot. But did you? You get back to your course a week later just to realize you remember next to nothing.

Does this story sound familiar? I've been there many times. I was working hard just to discover I forgot everything in a few days. The knowledge just didn't stick.

Is there a better way to learn if you've got only a few hours a week to spare? Yes, there is. It's all about building a habit.

Have you tried reading one page of a book a day? It takes just a few minutes, so it's not overwhelming. And when you start, you'll often discover one page is not enough. You get involved and want to know what will happen next. You'll decide to at least finish the chapter. So, when you build a habit of reading daily, you'll probably read more than one page a day.

You can use this same principle with programming.



To start your programming habit, schedule some minimal time for coding every day. Start small. It can be 10 minutes, or 15 or 30 – anything that feels manageable for you. Commit yourself to do it every day. It's crucial. One day a week is not enough. You can't go to the gym once a week and expect to improve your fitness drastically. You have to show up every day. It's the same with programming (and almost any other skill).

But what if you miss a day? Well, as long as it's just one day from time to time, it's not a problem. Try not to be too hard on yourself. Just make sure you won't skip two days in a row. After a few weeks, coding every day should feel natural.

But even if you learn daily, 10 or 15 minutes a day won't cut it. You need hundreds of hours to master a skill. With just 10 minutes a day, it will take years. How to overcome that?

To learn fast, you need to take advantage of the "one more page" effect I mentioned earlier. Don't have anything planned for the time after studying. The chances are when you start coding; you'll want to continue. You'll get involved in problem-solving or want to finish a feature, and ten minutes will naturally stretch to 30 or 40.

That's why it's crucial to have spare time after scheduled coding. Then if you get interested, you can continue. As a result, you'll study more than expected, remember information better and have a higher chance of avoiding burnout.

And you know what? You can do it multiple times a day. Let's say you have a whole Saturday just for you. Perfect time to do some coding. Even though you usually schedule 30 minutes a day for coding, this time, you feel the urge to binge straight 4 hours. Stop! Thirty minutes is enough for one sitting.

If you get stuck, you can always work on a problem a little longer. Don't force yourself to work for many hours unless you're in flow (and if you're, you won't be able to stop yourself, as the time will fly rapidly). Later you can take a walk, read a book, or do whatever you like. But promise yourself that after having some fun, you'll sit for another 30 minutes. Maybe another 30 in the evening. Splitting your learning into smaller chunks interrupted by relaxation will give you much better results.

“Schedule short learning sessions daily. Even 10-15 minutes will do.”

Coding isn’t enough to become a programmer. Your practice has to be effective. Too often, we don’t use our dedicated learning time wisely. We try to learn, but fail to do it efficiently, so we end up demotivated and tired. For your learning to be effective, it has to be planned, systematic, and focused on small, carefully selected steps that build up a skill over time. Let’s see how we can make our learning better.

STEP 1: PLAN YOUR LEARNING

You can’t reach any meaningful goal without a plan before you start learning. You need to know what skills are required and how to obtain them. Otherwise, you’ll jump from one seemingly exciting thing to the other.

How do you create a plan? Begin with reading or watching some tutorials about the industry. For example, if you want to learn front-end development, read some guides. It doesn’t require much time. You’ll soon notice everyone gives almost this same advice. For web development you have to learn things in that order:

HTML > **CSS** > **JavaScript** > **JS Framework you pick**

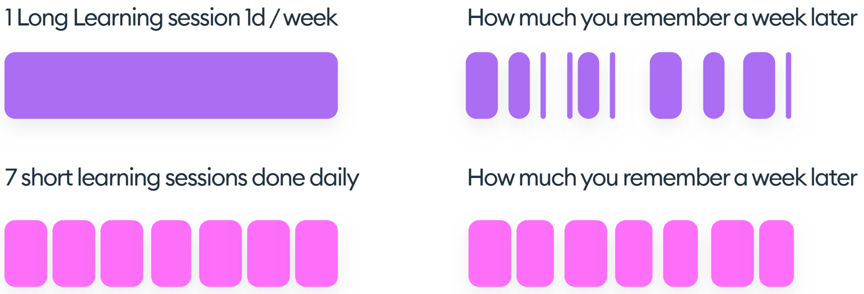
Now you’re ready to do some planning. Your goal is to set specific and realistic goals. Ignore everything besides the step you’re about to take. Focus on the HTML and decide how much time you want to devote to it. It will be a ballpark guess, as you don’t yet know how complicated the material is. But it’s OK. We’re just establishing some ground rules.

Let’s say you’ve decided to spend one hour a day for two weeks on HTML, and your goal is to create a well-structured website. It’s a great plan, and you’re ready to start learning. If, after a week, you feel you know enough, adjust your schedule, and get to CSS. Similarly, if after two weeks, you’re not yet comfortable, it’s fine. Change your strategy accordingly.

Always take it one step at a time and have a fairly detailed plan for the next week or two, as well as a bird's-eye view of your ultimate goal.

STEP 2: BE REGULAR

Practice won't help you much if you're not practicing regularly. Many people want to learn to code, but they say they've got only every second weekend to spare. Their thinking is like, "I'll spend two full days learning so that it will be 16 learning hours – on average more than an hour a day."


1 Long Learning session 1d / week
How much you remember a week later
7 short learning sessions done daily
How much you remember a week later

Learning doesn't work this way. In the days between your practice, you'll forget most of what you've learned. What's more, you can't efficiently learn for many hours straight.

You'll be too tired to benefit from the materials. There is also some evidence that sleep helps to solidify learning. You don't want to lose that advantage, so you should learn daily and rest often.

To sum it up, it's more efficient to learn an hour a day than seven hours once a week. Of course, if you've got more spare time on the weekends, use that time too. But try to devote some time to learning every day, even if on some days it's only 10-15 minutes.

Lack of focus is my #1 sin. When I was learning to program for the first time, I often spent up to five or six hours a day studying. I was watching dozens of video tutorials. After a few days, I remembered nothing, but I felt like I had put a lot of effort into learning.

Why wasn't my learning efficient? Because I wasn't concentrated. Many times, I was watching a tutorial while cooking or scrolling Facebook. Other times, I was so focused on re-typing the code that I didn't focus on the concepts presented or the course's goal. Many instructors made it even worse by encouraging me to code along with them. It's terrible advice, as most of us can't do two things simultaneously. So when we're typing, we're not thinking.

You can't fool yourself. The time you devote to learning should be all about learning.

Close all the unnecessary tabs in your browser, turn off all notifications, leave your phone in the other room. Focus solely on the task at hand. If you're watching a video, don't code along. Watch it, try to understand it, and then try to code by yourself. If you can't do something, try Googling it. Consider going back to the video as a last resort.

When you're reading a tutorial, don't copy and paste code. Try to understand the tutorial and create the code by yourself. The more you're focused on working on your own, the faster your progress will be. The tutorial should only be a guide, not a source of all the code.

Whenever possible, try to build things on your own. Tutorials can only get you so far. The real learning begins when you're in the wild. Little "for fun" projects can be a good way to mix up the learning experience next to the regular tutorials.

“Avoid distractions. Your learning time should be 100% focused in order to maximize your potential.”

GET FEEDBACK

To learn effectively, we have to know if we're making progress. We need external feedback. Someone or something telling us if what we're doing is any good. This way, we can build up the right habits and improve.

As programmers, we're in a comfortable position. Usually, we know we made a mistake because the code is not working as expected. So we can benefit from fast feedback loops and learn from our own errors.

But when we get some experience, we realize that the working code alone is not enough. We should take care of code readability, maintainability, performance, edge cases, and more.

How can we get feedback on that? We can look for programmers who could take and review our code. Maybe you know some experienced devs who can help already. Sometimes, experts offer code reviews for free on Twitter or other social media platforms. You can also look for programming groups and share your code there. Often, you'll find many people who are eager to give you some feedback.

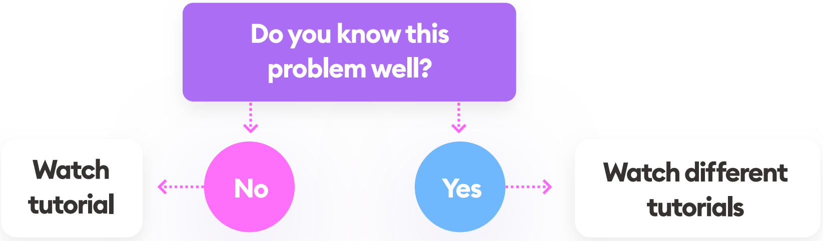
Don't dread judgment. Instead, strive to get feedback. I know harsh criticism hurts, but try not to take it personally. Treat remarks as a great way to grow as a programmer.

You can get feedback in more indirect ways. When you're solving algorithmic puzzles on sites like HackerRank or Codewars, look at other people's solutions. Analyze how they approached problems and in what ways your code is different.

You can also try implementing some popular libraries but always try to do so in your way. Later, compare your code and the library code. Seeing how proficient programmers solved the problems you tried to solve is a fun and effective way to be better.

You grow when you struggle. Initially, when you're learning to code, everything is difficult and complicated. After a few weeks of learning, some things will start to become trivial. Now you'll need to watch out for complacency.

You'll be tempted to watch tutorials on the same topic or do easy, repetitive tasks. I bet you'll even find some excuses like "This other tutorial will give me a different perspective on the problem." It's unlikely. Usually, you don't need a different perspective. What you need is to turn theory into practice.



You have to make a habit of challenging yourself.

I know how hard it is to challenge yourself. When I was learning, I often watched an hour-long tutorial, where I knew 95% of the things being shown. It was a waste of my time and gave me a sense of false accomplishment.

You may say that doing nothing is better than doing something. It's usually correct. But rewatching things you already know doesn't count as doing something. You'd be better off just going for a walk. Resting is more critical to quality learning than repeating these same simple things all over again.

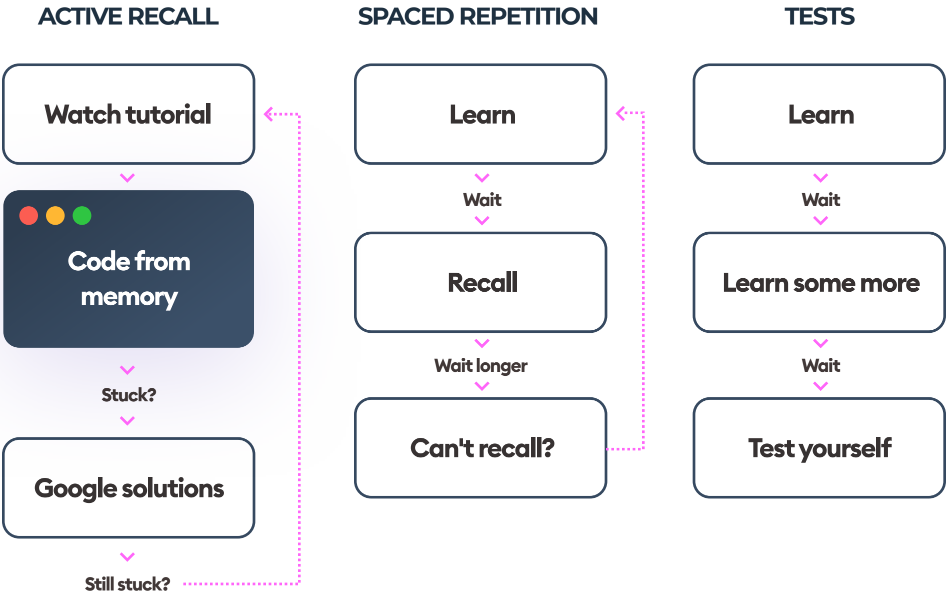
The other way we escape from challenges is by focusing on the easy parts of the task. If you're trying to learn JavaScript by creating an application, don't spend hours polishing CSS. Just use some CSS framework and push yourself to learn the hard parts.

We know how to build a programming habit. We know how to practice programming deliberately. Time to get into the details - get to know the efficient learning techniques.

When you've mastered coding fundamentals, the learning curve becomes steeper. New concepts become significantly harder to grasp.

Let's say you've seen a great tutorial. You worked along with your teacher for a few hours and built an excellent app. But after a week or two, you try to use concepts from the tutorial in other projects and you fail miserably. It looks as if you've forgotten everything. What to do now? You feel the urge to watch that great tutorial again. Don't!

Time is a precious resource. You shouldn't use it for rewatching or rereading. Here are three simple techniques to help you study effectively and avoid rewatching or rereading this same tutorial again and again.


ACTIVE RECALL
Watch tutorial
Code from memory
Stuck?
Google solutions
Still stuck?
SPACED REPETITION
Learn
Wait
Recall
Wait longer
Can't recall?
TESTS
Learn
Wait
Learn some more
Wait
Test yourself

> “Watching tutorials where you already know most of the thing only helps your ego, but not your learning.”

Active recall
Active recall is when you try to retrieve information from memory instead of rereading it. A typical example is the use of flashcards.

You can use active recall anytime when learning to code. Next time when you're watching or reading a tutorial, don't code. Focus on following a teacher, try to understand what problems they're trying to solve and what tools they uses. Then try to recreate code from memory. If you can't, search for the answer online. Only get back to the lesson if you can't solve a problem on your own.

Spaced repetition
You'll understand new concepts much better if you learn in smaller chunks spread over time than if you cram in one sitting. Create your system of repetitions, with increasingly longer breaks between each learning session.

For example, learn about closures for thirty minutes. Check out what the closure is and when to use it, see some code samples, take notes, or make flashcards. Tomorrow, try to recall information about closure from memory. If you succeed, try again after three days. Each time lengthen the delay between repetitions. If you fail – stick to this same delay.
Experiment with different schedules to find the one that suits you the best.

Tests
Remember how you used to learn at school? You may have studied over the semester, but you worked hardest and learned the most before the exam. Use it to your advantage. Create a simple schedule. For example, devote two hours to testing once a month. You can easily find tests online or create them by yourself.

The exam doesn't need to have a traditional form. It's even better if you test yourself by making an app using the knowledge you want to solidify.

Writing requires thinking critically and organizing your knowledge. You'll be amazed at how swiftly you learn when you have to write down a new idea. By writing, I don't mean taking notes. Notes are a useful learning tool, but you'll grow faster if you try to explain programming concepts and techniques to others. So try creating a blog or other type of course for fellow developers.

Not starting a blog when I was a junior developer is my greatest regret. I could have accelerated my education and helped others. I know what you're thinking: "I'm not experienced enough to teach anyone anything. Who would ever read what I wrote?". I'm sorry, but you're wrong.
You know something about coding, and what's even better, you're not an expert. Experts often know too much to help beginners effectively. They forgot what was difficult for them a decade ago, assume readers have vast knowledge, etc. As a beginner with some experience, you're in a position to help others.

And what will happen if no one reads your blog? Nothing. But the knowledge you gained in the process will stay with you.
To write a blog post, you need to think the topic through, explain in simple terms, and structure your mental models. Often you believe you understand the subject. But when you start writing, you realize it's not the case. You need to clarify and enrich your understanding. By the time you finish the article, you'll grasp the subject.

So if anyone reads your blog is not relevant. What is relevant is your growth. Writing a blog may seem underwhelming. You are a programmer, not a writer. If you're worried you can't write, try advice from the beginning of this chapter. Create a habit of starting slowly. Even one sentence or paragraph a day, it's enough to start. After a few weeks of writing, you'll become more confident and see the benefits of writing. Use that to write even more, and accelerate your career.

Key takeaways

- ✓ learning to learn is the most important skill of a developer
- ✓ building a learning habit will help you to skyrocket your career
- ✓ you learn by practicing deliberately; most other things are just a distraction
- ✓ try challenging yourself to get most of your learning time
- ✓ use the learning techniques that suit you
- ✓ teaching others is essential to solidify your knowledge

18.

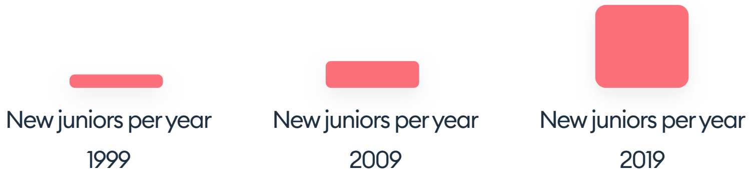
Learn something rare

Learn something rare

BE UNIQUE

Developers are in demand, and it doesn't look like it is going to change anytime soon. The whole world still lacks many specialists across different branches of the tech world. From our perspective, this sounds perfect, doesn't it? Indeed, it can be a great incentive to learn to program and start a career with a stable position and good prospects for the future. It looks like a dream come true for all the people who would like to do some coding.

This comfortable and attractive environment comes with a price. Everywhere where demand is such high, after a bit of time, comes equally great supply. Development is no exception to that rule. All the reasons mentioned above are valid for you, me and for thousands of other aspiring programmers trying to cut their share of this glorious cake. There are lots of people coming to development from different backgrounds every day. One of the main reasons for that is programming is a field that is essentially based on so-called **hard-skills**. These are skills that one can acquire by their own effort to a good measure.



What it means for us is that the competition grows higher, and that's even more true for entry-level positions like internships or junior software developers. You may question yourself, how can you improve your chances of getting noticed, leading to more interviews, and finally landing a job. How to stand out from an increasing crowd of juniors? Frontend Unicorn team to the rescue!

Frontend Development has more opportunities to expand your knowledge and skill set than you probably imagine. It is worth exploring them for multiple reasons. The obvious one is that you can impress your future employer by presenting a skill that makes you stand out from the crowd.

And I'm not talking about being able to lick your elbow. However impressive, it's probably not something that would give you an advantage during the software engineer interview. I'm talking about skills that are closely related to your field of work and can possibly show that you are open-minded and ready to make use of your current knowledge to solve different problems.

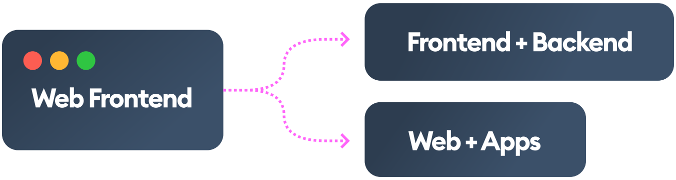
If you are more of a freelancer and don't need to impress anyone to get employed, you still probably need to make yourself stand out to get potential clients' attention. Improving the quality of your outputs may be the first thing that comes to your mind. But in most cases, you are not the only one in that field, and putting quality work out there may not be enough. The same rules apply, and unique skills make you stand out and help with marketing your personal brand.

Even if you don't need any more attention and are fine with your job as an experienced developer, you should never stop learning and broaden your field of expertise. Some of the skills I'm going to mention could make a nice side gig, giving you an additional solid source of revenue. Some of them can become a creative outlet that would improve your wellbeing.

Programming, in my opinion, can be rightfully called a creative work in itself. But quite often, the output is limited by business needs and a feature roadmap. I want to show you how you can use your current skills to step out of that zone and try to build something different. You can go so much further with your knowledge of HTML and JavaScript than just making websites and apps.

> “Try to stand out by expanding your skills into less popular or more difficult directions.”

If you are going to learn something new, why not start with something that may happen to be the most useful in your job. If you are a frontend developer, you can follow two different paths when expanding your skills. You can either stay in the "web zone" and go for a full-stack experience or stay more in the "frontend" side and try working on mobile apps. Both of them are viable routes for a JavaScript developer, with a full-stack approach being probably a more common one.



Especially if you are an experienced developer already, being able to create a backend for your web app will make you self sufficient and not only limit you to the front side of the project. Now you could handle the whole project by yourself or at least be flexible enough to get involved in either of them, depending on your client requirements. With your current skills, the natural entry point would be a NodeJS environment, which lets you use the language you already know (JavaScript) and run it on the server. We'll talk a bit more on that subject in a different chapter of this book.

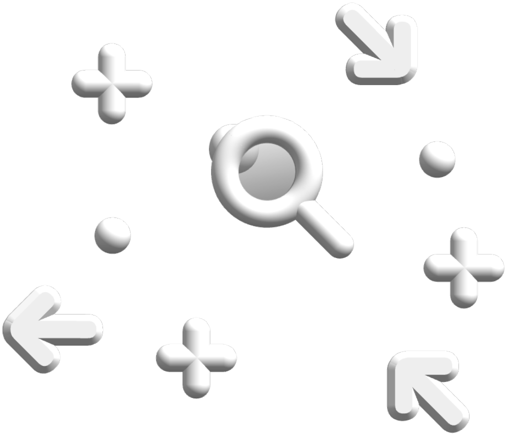
Going for mobile development might not be the first thing coming to your mind when thinking of broadening your frontend developer skillset, but it's definitely worth a shot. There was no better time before to do so, than now. With React Native growing in popularity and being proven as a production-grade solution for almost any mobile app project, you can start putting apps to the Google Play Store or the App Store in no time. Learning mobile app development may be the way for you to craft the whole frontend experience of the product you are building, both for Web and mobile devices. If this sounds interesting to you, we decided to cover this topic with more details in another chapter too.

Here is another approach to learning something new but still viable as a part of a frontend developer job. I'd even say that it can be a career on its own. At first, data visualization doesn't seem like a field that could benefit from your knowledge and skills. But when you understand that it's main purpose is to present data in a most friendly and understandable way, you may see that this can be achieved in numerous ways, like making it animated or adding an interactive user interface.

What are we using our JavaScript for, if not to display some data that we got from the database or to make a basic HTML website more interactive? If you think about this way, you can see how those fields can overlap. Maybe you even added a simple line chart to your app before. Now you may think that for sure, there have to be more specialized tools, and that's correct, but what programming gives us is almost unlimited freedom of creation. You can model and prepare data however you want and come up with lots of creative ways to present it to the world. And recently, JavaScript became a popular tool to do so.

One of the best libraries for that is D3.js, and I can not recommend it enough and encourage you to at least take a look at this one. It comes with a learning curve, and it's rather a steep one. But when you get a grasp of it, you will see it's almost unlimited possibilities. It's fascinating on its own. If you've already been working with some charts in your project, there is a great chance that the library you used is based on D3.js
You can work on data visualizations alone, as a freelancer cooperating with magazines, or you can incorporate your freshly acquired skills into your next fronted project. Granted, it is going to include some kind of a dashboard or charts. **D3.js** gives you a massive advantage in terms of designing a custom-tailored solution and not relying on lacking third party integrations. This one alone should make you a more favorable candidate for projects in the fintech or data analytics sectors.

Yeah, 3D like in three-dimensions. 3D used to be a thing you did only with dedicated, advanced software like Blender or similar. But now browsers are making better use of your computer's power than ever before, giving you, a frontend developer, even more possibilities. With just a bit of JavaScript and an HTML canvas, you can run a three-dimensional scene in your browser without having to learn all those complex tools.



One of the best entry points to the current state of 3D Web is by Three.js. It's your go-to library if you want to spin up a 3D scene in your project. You can load your custom models right there or create them from scratch using code. It's a full-fledged 3D environment with textures, cameras, lighting, and all the exciting stuff you need to start your journey. Ever wanted to create a better game with JavaScript than a simple Snake, Pong, or Tic-Tac-Toe? Three.js and similar libraries are what you need.

This one may be even more niche than data visualizations but still can be a good selling point on your resume. You can make outstanding landing pages or work on specialized tools that make use of 3D views in their products with skills like that.

“Data-visualization or 3D are a great way to stand out from the crowd.”

Remember how I said that programming is, in my opinion, a creative act on its own, but the needs of the business often limit it? Now, I'd like to show you how you can utilize everything you learned to create something different than another app or website.

JavaScript is a fantastic way to get into a field called "Creative Coding." What hides behind this term is using programming to create something expressive, in any form that you like. At this point, you should see where I'm coming from. D3.js visualizations or 3D scenes with Three.js may happen to be tools that you use at work, but nothing stops you from using them to create art.

There, I said it.

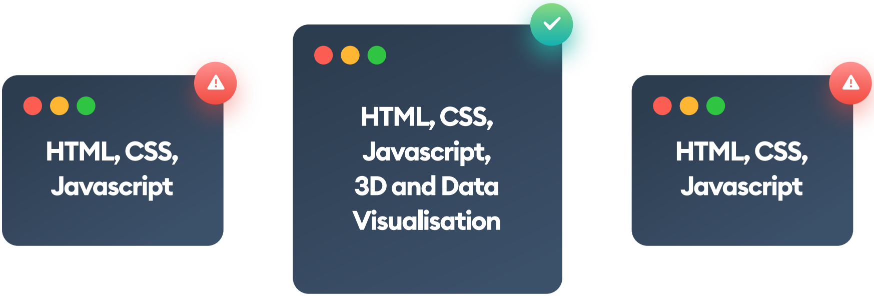
You can create art with JavaScript. Perhaps not *in*JavaScript itself ;)

Creative Coding is all about freedom in programming and making something fun. Tools I've already mentioned should be a great starting point if you are already familiar with them to some extent. But if you are just starting out and want to give creative coding a shot, you may want to use something with a higher level of abstraction that hides away from your sight all the non-important details. P5.js is precisely that. It is all about making interactive experiences in a very convenient manner.

With Creative Coding, you may not find it helpful during your day to day job, but rather make it an excellent outlet for your internal artist. It may not help you get another web app project but can become your little side gig's driving motor. Who said you couldn't sell some prints with outcomes of your JavaScript? Use a pen plotter on fancy paper, and if it's good, you may want to get a booth at the next fair and try your luck selling them.

Keep an open mind and explore, see what people are building and what makes you excited. That's the one thing I'd love you to learn from this chapter.

Maybe you like machine learning? It's worth looking at the JavaScript version of TensorFlow. You can mix and match your skills to fit what you like. It doesn't have to be such an extensive topic as those mentioned above. You can go for something small. Maybe you enjoy working with maps, and you want to be good at using them in your projects.


HTML, CSS, Javascript
HTML, CSS, Javascript, 3D and Data Visualisation
HTML, CSS, Javascript

Having a skill or two that sets you apart from every other junior developer is essential for success in the industry.

Having such a "thing" within your coding experience is an excellent way to stand out from the masses of regular developers. Most of the juniors entering the job market each year have the same or similar skills backed by the same or very similar projects. You will have something special to show your future employer or client. Or at least something new you can talk about with passion, which is always a plus.

Key takeaways

- ✓ here are many developers trying to get the same job as you. Make yourself stand out by learning something niche and fascinating.
- ✓ ou can use your web development skills for something more than websites and apps - be creative!
- ✓ Data visualization, 3D rendering, or creative coding are fields worth looking at if you want to create using code.
- ✓ It doesn't have to be something big. Mastering a specific tool is equally good.
- ✓ Keep your mind open and explore. If you find a subject you are passionate about, and it helps you grow as a developer - it's a win-win.

19.

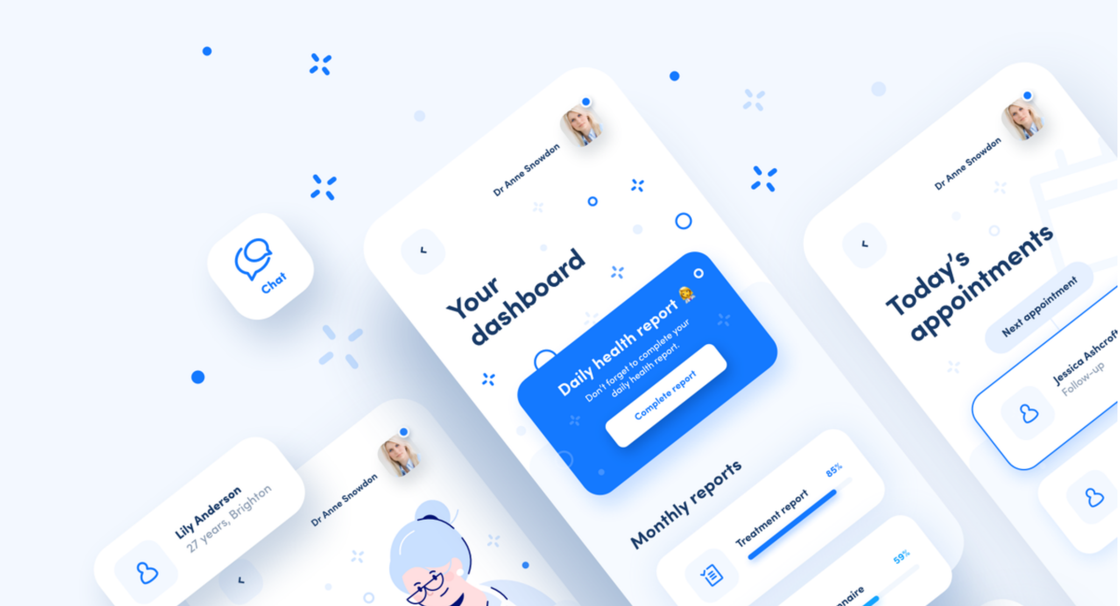
The mobile world

The mobile world

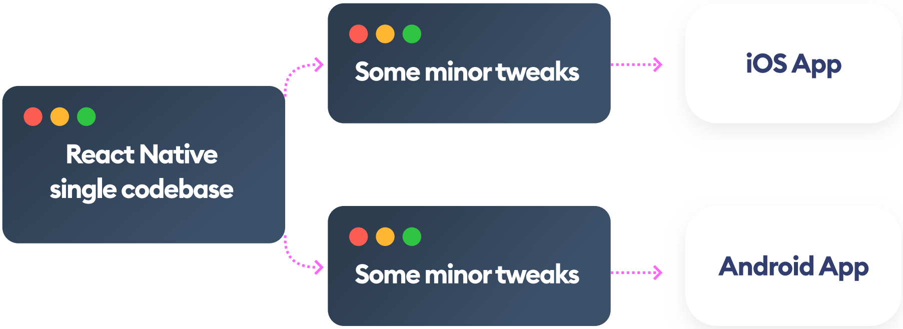
DON'T BE AFRAID OF GOING MOBILE

Since, we're talking about learning something different than your regular web development, I think we got one more thing worth exploring. As a frontend developer, you don't have to limit yourself only to the **web** frontends. There are other types of applications that you can create while utilizing your skills.

Mobile development is one such a field. You might be a little confused now. It's perfectly understandable. You may think *Mobile development? Aren't mobile apps made using Swift or Java, or Kotlin? I don't know those languages; I'm just a JavaScript developer at best, and I have plenty to learn here. I don't need another language or two, and there are probably some frameworks involved as well. Well, that's true, but let me explain before you toss this book in the corner, or more likely the device you are reading this on, thinking that we don't know what we're talking about. We do, trust me.


Chat
Lily Anderson
27 years, Brighton
Dr Anne Snowdon
Your dashboard
Daily health report
Don't forget to complete your daily health report.
Complete report
Monthly reports
Treatment report
85%
59%
Dr Anne Snowdon
Today's appointments
Next appointment
Jessica Ashcroft
Follow-up

The answer is relatively straightforward - React Native. It's probably the most mature of all the frameworks you can use to develop mobile apps using Javascript tooling. If you're already familiar with React, going for a ride with the Native brother might be fairly accessible and painless. All the rules you already know can be applied here as well, with some minor changes here and there.



For example, you don't use *divs* in your JSX here, we got *views*. But they're used in a similar manner. You can still use Redux or Context if you like; almost all those standard JS libraries you are used to working with are at your disposal. You don't use CSS directly, but the styling is made to look and feel like CSS. There are a few more differences like that, but you get the gist. Now you may ask, * That's it? Just a few little changes and my React app becomes an iOS or Android app?* And my answer is - almost.

You'll still have to learn how each platform works, how it is configured, how you can debug your apps or why navigation works differently. But for the most part, you can even skip using the dedicated code editors like Xcode for iOS or Android Studio for well, Android. It might sound a bit less straightforward than I promised before, but as a frontend developer working with all the complexities of Webpack, Linters, transpilers, etc. is it really that much harder? I don't think so.

“React Native is one of the best paths to take from web to mobile development.”

There is one thing about this chapter that I don't like. We've been talking about learning something different in the previous chapter, expanding your horizons, getting out of your comfort zone. I was trying to encourage you to be brave and bold, like a true coaching mentor, sort of. And now we're ensuring you that you can be a mobile app maestro with almost all the skills you have and maybe just a sprinkle of new knowledge that you shouldn't be too scared of learning. You know what? I got something even better. And that's the thing worth looking at if you truly want to go all in.

Leave the JavaScript world behind you. Or don't, you don't have to burn the bridges, you can do both. But what I'm trying to say is, if you are going to try building some mobile apps, maybe try doing it with the languages and frameworks they were meant to be created with. I would not make such a recommendation a few years before. Things have changed since then, and I think that proper native mobile development is more accessible than ever before. Why is that?

Languages have changed, and so did frameworks. One thing is that Swift is more of a friendly language than Objective-C (the primary language for iOS development) was. But this alone is still not something that would make me recommend this to the frontend developer so frivolously. Swift is a big step up, but the way UIKit and Autolayout (think of it as UI framework for iOS) work is different from what you already know. To the point that it'd be quite a bit of an investment to learn all that while trying to kickstart your web developer career.

SwiftUI is the answer. It's a framework that can be a replacement or a fantastic companion for the UIKit. But SwiftUI's major advantage is the **declarative** style of defining the interface in code. And this might be similar to the way you've been writing your HTML or JSX. In a way, you know, we are talking about a completely different language that has its own quirks and features. Still, overall, SwiftUI is definitely more comprehensible for a web developer like you.

Now, you may say that I'm only talking about Swift and iOS, and what about the Android? That's because I assume that you are at least familiar with the HTML and maybe even React or Vue, and based on that skillset, I think that iOS, with its current state and the SwiftUI, is the more beginner friendly of those two.

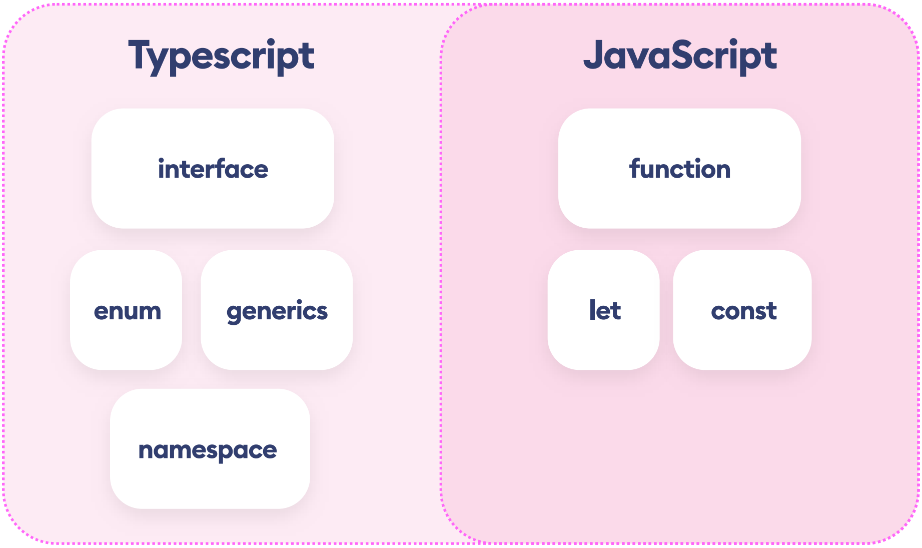
If you are not the type of person to be discouraged by that, you can try Android first. But for the sake of your sanity, don't go for both of them at the beginning. Try focusing on one of those platforms and understand it until you feel comfortable using it. It's is going to immensely help you when switching to the other one in the future. That's because they are not that different in their core concepts, and if you understand how the mobile world is spinning, you'll be fine. The implementation details are what differs enough to make you confused if you'd tried to understand both of those concepts at once.

When it comes to Android, the developer experience has considerably improved since the Kotlin language introduction. Nowadays, Kotlin is a first class citizen at the Android framework and you can use it for all your projects, instead of Java. If you happen to know Java already, because someone has recommended it to you as one of the first languages to learn, because of unlimited job opportunities blah blah... Or maybe you had to learn it at some point in your formal education, you're set.

You can still use Java for your Android projects, but keep in mind that Kotlin is probably the future of this platform, even Google itself is using Kotlin for their apps now. Personal preferences aside, Kotlin is also more of a loved language than Java among developers, and it's meant to solve many of Java's issues. The other cross-platform advantage of this language is that its syntax is quite similar to Swift's. This could help you if you'd ever like to get familiar with iOS in the future. The same goes for Swift developers. If you know it, you should be able to understand the Kotlin code without issues.

Now I'd like to sell you on the benefits of you putting in the hard work and learning languages that are not JavaScript. They can help you understand TypeScript better.

TypeScript is a superset of JavaScript, and its primary goal is to add some static typing to the language. You see, languages like Swift or Kotlin were developed with static typing in mind. You can't go for , "non-strict" mode with them like in TypeScript, where you don't have to adhere to the static typing through your whole codebase.

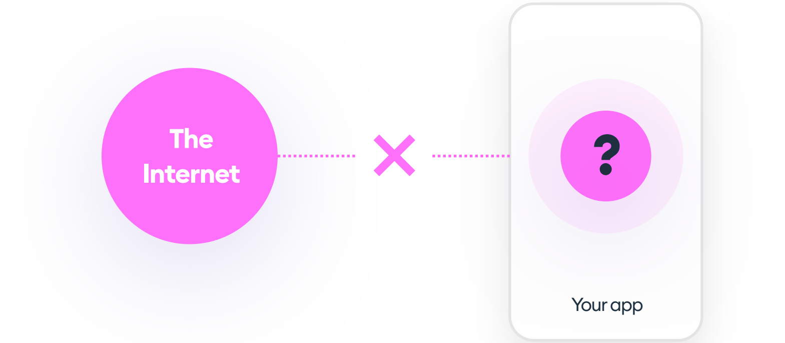


It's a common thing I notice when working with developers that were only using JavaScript before trying TypeScript. They don't fully understand what benefits come with static typing and how they should use it to make their life easier. Statically typed languages are the remedy.

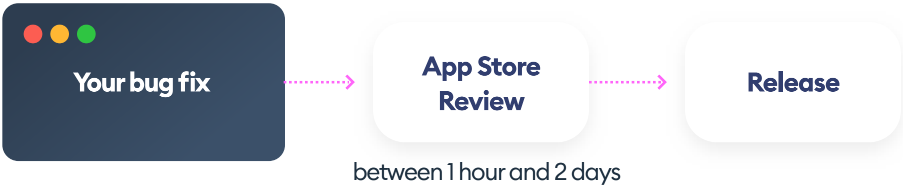
You should try one of those languages just because it might help you be a better web developer. Learning mobile development might help you with your web-related tools. Isn't that cool?

> “Learning mobile development might help you with your web-related tools.”

Another advantage of exploring other technologies, like mobile apps is that you will face completely different challenges during development. Those apps work differently from your website, and you might want your app to work offline. It's their default behavior if you think about it. What would you say if your Mushroom-Pokedex app was not available when you are in a forest and you lost your internet connection. How would you solve that?



One thing that differs a lot is an application release cycle. You probably didn't have to worry about that while working on a website. If there was a bug, you could apply those changes fairly quickly. No one would stop you, apart from your expertise in that matter.



When it comes to mobile applications, every release has to go through the App Store or Google Play Store. Sometimes it has to be reviewed, and it could take up to a few days. This makes a significant impact on how you code and how you test your app when you know that you won't be able to fix that typo, and it's going to stay as it is for a day or two.

Improving your skills is a great thing, but let's be honest, mobile development **IS** fun. Smartphones or tablets come packed with cool tech and features that you can get under your command with code. It's like a playground with toys that you couldn't use before with your browser API.

For example, Augmented Reality in a browser is a thing, but it's barely useful compared to the native libraries and APIs that systems like Android or iOS give you. The same thing goes for Bluetooth or Camera frameworks. You can get really creative with these.

But for the most part, you should explore and see what makes you excited, and there are many opportunities for that in the mobile world. Who knows, maybe you will send your next resume to a Junior iOS / Android Developer position? If not, that's cool, but now you would know that this is something you don't want to do, and this knowledge is as valuable, if not more.

Key takeaways

- ✓ Mobile development is now more accessible than ever. You don't need even need to learn other languages. You can start with JavaScript...
- ✓ ...but we encourage you to go full native. Learning a new language like Swift or Kotlin may help you understand TypeScript better. Not to mention, it's going to look great on your resume.
- ✓ By the way, SwiftUI is a framework for you if you want to keep it as similar to React as possible
- ✓ Mobile app development presents a bunch of new challenges that you are not going to face with web apps, like a longer deployment time or working offline

20.

Impostor syndrome

Impostor syndrome

HOW TO FIGHT IT?

Do you feel like a fraud? Do you feel not good enough, not smart enough? If not, you probably will at some point.

You'll find your first job, someone will trust you to do it well, and all of a sudden, you'll feel inadequate. Like you don't belong. Simply speaking, many times, you'll be out of your depth. As Neil Gaiman put it:

"The first problem of any kind of even limited success is the unshakable conviction that you are getting away with something and that any moment now they will discover you. It's Impostor Syndrome, something my wife Amanda christened the Fraud Police."

Am I good enough?
Was it just luck?

So, what you need to do when the impostor syndrome strucks you? Be happy. It means you achieved something. Congratulations!

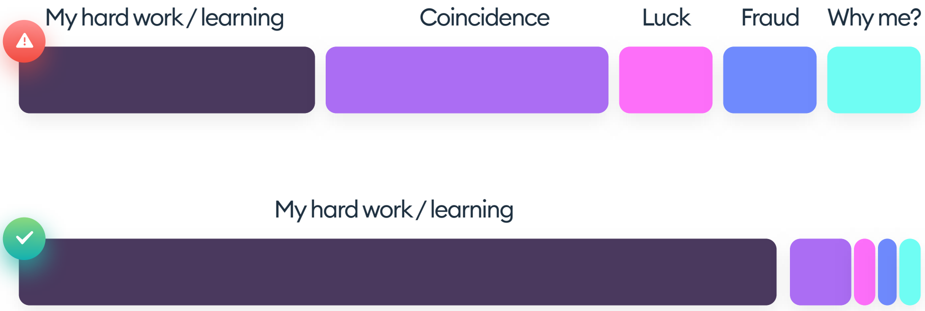
But I bet it's not enough. The creeping feeling of dancing on a knife's edge and being just a misstep from failure is typical for self-learned programmers. So let me try to help you with it.

The tactics on the following pages will help you overcome the fear of failure and thrive in the software industry.

ACCEPT IMPOSTOR SYNDROME FOR WHAT IT IS

The first step of healing is accepting your problem. You need to acknowledge what you're feeling and why. So observe your feelings. Try not to be judgmental; just examine what exactly you're thinking.

When you've examined your thoughts, consider if they're helping you. Do you feel motivated to learn more or just frightened and passive? If these thoughts aren't beneficial to you, maybe you should let them pass?



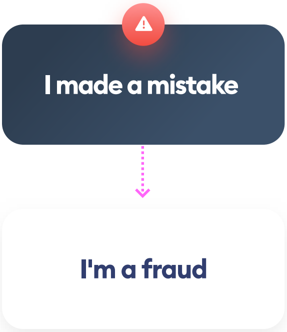
Try to differentiate between internal (MY hard work) and external issues (luck, fraud etc) and always try to make your effort more valuable than these doubts.

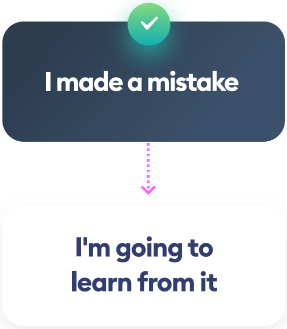
Be kind to yourself. You've gone a long way to get a job, that's worth something. No one is an expert on day one. Learning and growing to your role takes time and effort. You'll get there one step at a time.

And it's expected you've got impostor syndrome; we all struggle with it. It doesn't mean you're stupid or misfit; it means you're a human. It also means you're not complacent. It's a good sign, you're much more likely to grow when you feel like a fraud than when you're overconfident.

Don't look at Twitter and other social media too much. Most of the programmers who inhabit Twitter are perfect; they never make mistakes. Actually, one of them had their last bug in the late nineties. I don't know how so much of the software is buggy, almost to the point of useless, if everyone is such a proficient programmer.

Just kidding. Of course, programmers' public image is something different from their actual skills. On the surface, they may seem flawless, but they often make mistakes - just like you and me.



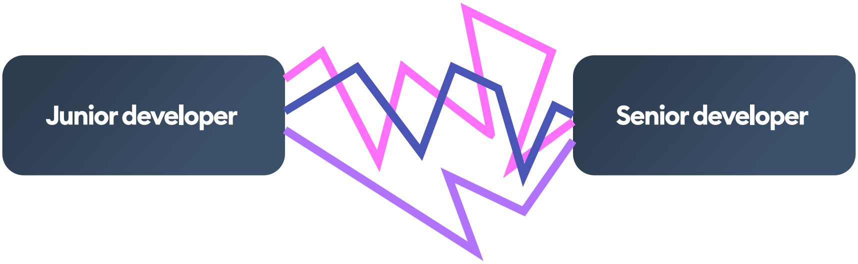


In software development, nobody knows what they're doing. Almost every startup fails; the most prominent companies routinely ship code that doesn't work, and no one wants to use it. Failure is a standard, not an outlier. You're going to make mistakes and fail like everyone else. If anyone tells you he's walking from success to success without failure in between; he's a liar.

There is one more thing. You'll be wrong many times. But being wrong doesn't make you a fraud. Everyone in every field missteps - be it a sportsman, an entrepreneur, or a developer. The way we respond to our failures is more important than the fact we've failed. So embrace your mistakes and learn from them.

> “In software development, nobody *really* knows what they're doing.”

Negative self-talk is terrible for you. You need to believe in yourself to grow. But the negativity comes from somewhere. It may be true that you lack many skills and don't know many things yet. We all do. What you need are humility and honesty - you can't and won't ever know everything. But you need to be able to learn fast and patch your knowledge gaps.



There are many paths of growing as a developer - there is no "right" one. You just need to find yours and believe in yourself.

A considerable part of this book covers the things you need to learn and how to learn them. Take advantage of it. Learn from other places, find different perspectives, try what works for you and what doesn't. You'll always find someone with more knowledge, but that's fine. All that is required from you is the ability and willingnes to learn and become a better developer.

So use impostor syndrome as a tool to develop yourself. Every time you feel inadequate try to learn something new. Something that will make you better, closer to your goals.

REMIND YOURSELF WHAT YOU ALREADY ACHIEVED

When you're low, it's good to remind yourself how far you've already come. Track your successes somewhere, and get back to them whenever you need. You'll be amazed at how much you achieved since you started learning.

You should also take pride in how your colleagues treat you. Do you get more challenging tasks? Are you expected to be more self-reliant? These are indications that your team believes you're getting better. They feel they can trust you.

There are other methods to see how much you've developed. Update your CV with the skills you gained. Maybe you've learned a new language or framework? Or got experience with project management or version control tool? Put all of it on your CV, not only it will be more up to date, but you'll also see how you've grown.

A great way to see how far you've gone is to look at your old projects' code. Take a peek at your applications from a year before, and you're likely to have dozens of "how I could have done that" moments. You'll see where your mental models failed, how many edge cases you missed. And that's great! It shows you've grown and you know a lot more than before.

If you've got some time, you can event refactor one of your old projects. Try different patterns and solutions; see how much you can squeeze off that app. This kind of experiment with code is a phenomenal learning tool and a confidence booster.

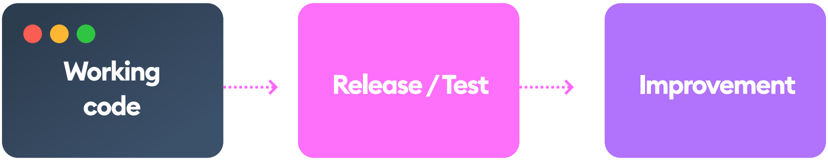
Remember, you're not racing with anyone but yourself. So that's not a problem if you're getting better slowly. The important thing is you're becoming better.

WATCH OUT FOR PERFECTIONISM

Perfectionism is a killer for productivity. I'm aware of the tension between getting things done and a quality of those things. But as a junior developer, your most important goal is to grow. And you can't grow when you're petrified out of fear of making something wrong.

Also, perfectionism is often procrastination in disguise. You are afraid of making something imperfect, so you postpone the real work to protect your ego.

In some cases, solving the problem quicker, may reveal a bigger issue with the entire project. It means that if you did it quickly, you may realise that you need to start over a lot sooner. With the perfectionist approach, at this point you'd still be tweaking code that will turn out to be useless.



So get your hands dirty. Experiment, try to solve the problems at hand. You'll worry about making code great later; your first goal should always be solving a problem. It will give you insights and practice necessary to polish your solution later.

> “Keep a list of what you've already achieved and look at it whenever impostor syndrome kicks in.”

Impostor syndrome is nothing to be ashamed of. You should talk with your manager about your doubts. You're likely to get some advice or review of your responsibilities; after all, there may be too much on your plate. You should not suffer in silence, and in this situation, honesty will pay off.

Also, a conversation with the manager will probably give you some feedback. Maybe there are some areas you actually should improve. Or your manager thinks you're great. Even if you disagree with that positive review and are convinced you're an imposter, I've got good news - at least you managed to fool some smart people that you're not.

But all jokes aside, as long as you're getting the job done and still learning new skills in the process, the impostor syndrome should start to slowly fade away. Try not to think about it, instead focusing on the task at hand.

Key takeaways

- ✓ impostor syndrome is real but can be used to your advantage
- ✓ take pride in your accomplishments, never forget where you started and how far you've gone
- ✓ perfectionism can harm your work and self-esteem, be aware and watch out for it
- ✓ don't be shy to talk about your problems with your boss

21.

Details that make a difference

Details that make a difference

MINIMUM VISIBLE PRECISION?

Normally, MVP means a minimum viable product - the most basic version of the product that is necessary for testing if it makes business sense. For years, analyzing coded designs we jokingly used the Minimum Visible Precision moniker as well. Many coded interfaces, both web and mobile ones, are coded with the premise of: it displays on the screen, so it's fine.

This is a wrong way to look at what we do. Sometimes, with very little additional work, we can improve the usability of the website or application that we are working on. These improvements can take it from a "so-so" product, into a polished, fun to use experience. And because most people buy with their eyes, that first impression is very important.

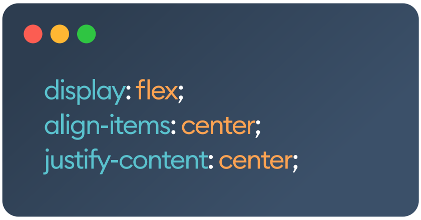
Many of these problems are not fully visible at first glance, but they facilitate better interaction with all the elements on the page. Such details in the eyes of perfectionists increase the value of the product.

And even in the eyes of regular users, they often subconsciously feel something is done "right" in the product. And they reward you with their attention, and hopefully, their wallets.

In this chapter I'll cover the most important problems that often have very quick and easy solutions and in the end they do make a difference. For better understanding of some of these practices, it's best if you first read the design chapters of this book.

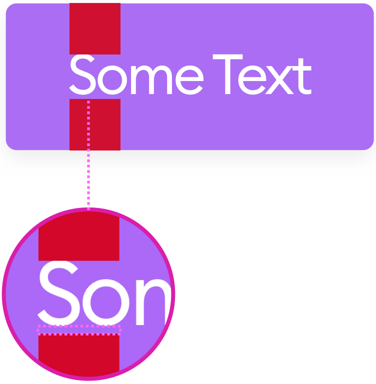
The easiest way to quickly see the level of precision a project has, is to check its buttons. It's literally one of the biggest problems in both design and front-end at the moment. The main issue is the vertical centering of the label inside the button. Most websites and apps fail to do this right, but it's understandable. The left and right margins can also be uneven, and you can use the same technique to test for that - but from our experiences the main problem is almost always with top and bottom padding.

If you go with:



```
display: flex;
align-items: center;
justify-content: center;
```

You may assume that it's enough. But in reality when you look at buttons like that you can quickly find out that the font can still be too high or too low. Here's an example of what it looks like. Designers can use a red-square technique to quickly see how well spaced the actual font is - and not its bounding boxes.



As you can see, in the example on the left, there's two extra pixels under the text, even though the object is centerted. In such a case it's best to align it by hand. A good rule to start with is to have the button and font-height as both odd or even. If one is odd and the other is even, it'll be impossible to align it vertically.

CLICKABLE AREA

According to WCAG, the minimum size of the clickable field is 44x 44px. Think how many times you have had contact with a smaller field? How many times has it been hard to click on an item because it has a tiny clickable area?

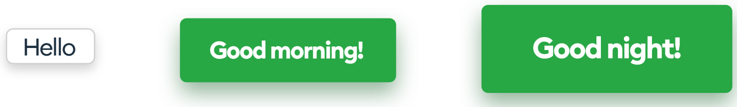
The design won't tell you anything about the clickable area, because we already enter the knowledge of UX. So if a button is small, we may assume that its clickable area is simply the area of the button itself. We are all users of the Internet and various applications, so we all know what annoys us most.

BUTTON

We should use the <button> element when we want users to click on it. For all non-buttons, we should add ARIA attribute`role = "button" to force them to behave like a button. This role allows screen readers to treat it as a button. Read more about accessibility here - Accessibility.

Intro for the role of buttons according to MDN:

“Adding role=“button” will make an element appear as a button control to a screen reader. This role can be used in combination with the aria-pressed attribute to create toggle buttons.”

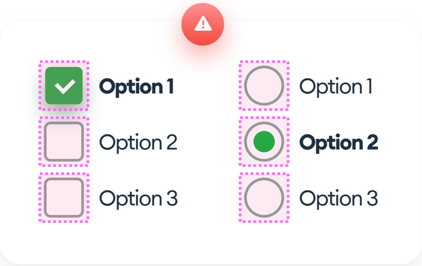


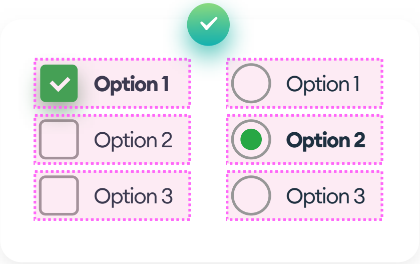
Starting from the left side, the first button is the default rendered by Chrome on macOS. The middle button got the background colour and padding. It is immediately much more visible than the first button. But we can also improve this one by increasing the font-size and padding to be 44px high

> “Make sure your buttons are big enough for a comfortable click and tap.”

If we have a radio button or checkbox, the user assumes that they can click on the label to activate/deactivate.

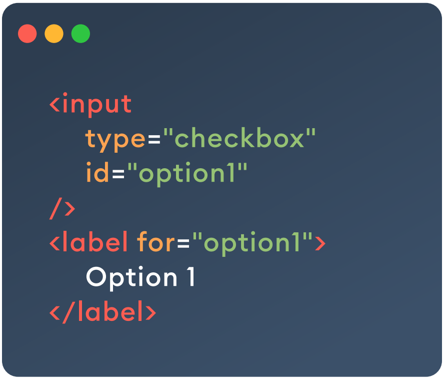
In the picture below, the clickable area is limited to the checkbox and radio button only. In other words, you cannot click the label to activate/deactivate.


☑ **Option 1**
☐ Option 2
☐ Option 3
○ Option 1
● **Option 2**
○ Option 3

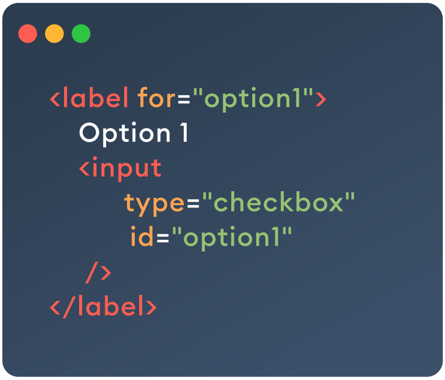

☑ **Option 1**
☐ Option 2
☐ Option 3
○ Option 1
● **Option 2**
○ Option 3

☐ = clickable area

This is bad from a UX perspective. It makes the area the user has to click much smaller, and their first few clicks can be unsuccessful, leading to frustration with the entire product. However, it is easy to solve by adding a “for” attribute to the labels (left). Or, alternatively, you can also put an input inside a label (right).

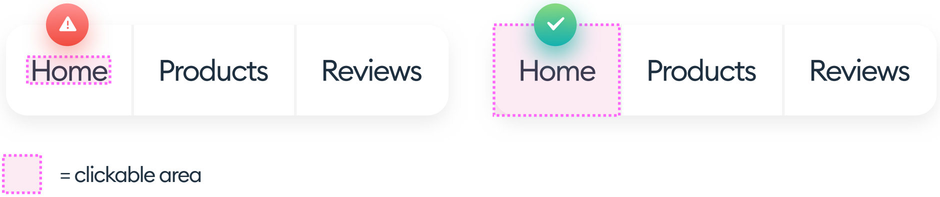


```
<input
   type="checkbox"
   id="option1"
/>
<label for="option1">
   Option 1
</label>
```

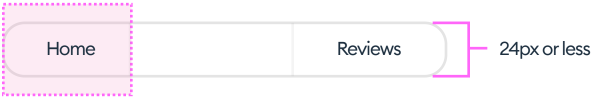


```
<label for="option1">
   Option 1
   <input
       type="checkbox"
        id="option1"
    />
</label>
```

While creating links in the menus or sidebars, you need to remember that the link should not be the size of the text alone. The best way is to either use the "visual" background size as a reference for the link size. A little padding around the <a> element will do the trick.



In case of very small elements you can even go a little outside its bounds. It works well especially on mobile devices, as it makes it a lot easier to access elements that are perceivably small.

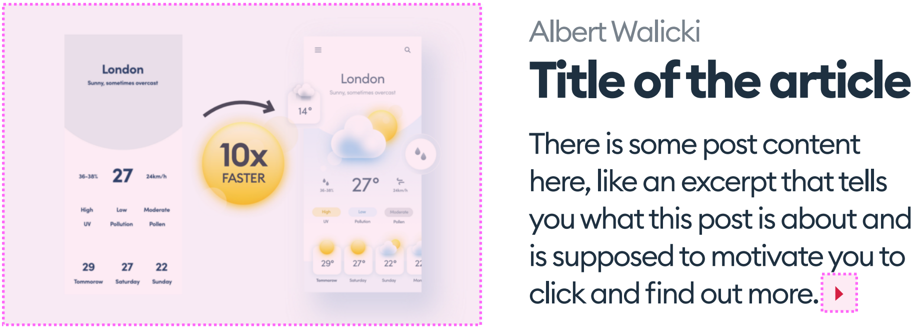


If your clickable element is smaller than 24 pixels, it's best to expand the clickable area to at lest 32/40px, making an invisible button. Trust me, your users will thank you!

If you're making responsive versions of these controls, keep in mind that 44px is the minimum clickable area size for mobile. This is important especially now, when most web traffic is actually coming from mobile devices.

ARTICLE SECTION

Article sections can be tricky, but with a couple of useful tips, you can make them work in no time. We worked on a website that utilized this kind of an article section.


London
Sunny, sometimes overcast
27
10x
FASTER
London
Sunny, sometimes overcast
14°
27°
29 27 22
Tommorow Saturday Sunday

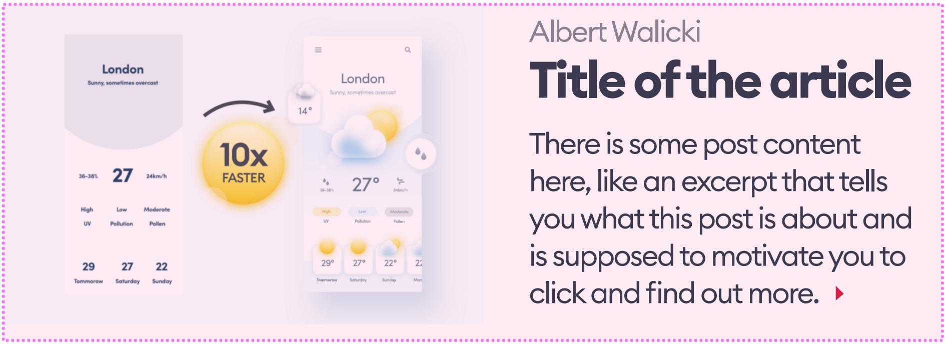
Albert Walicki

Title of the article

There is some post content here, like an excerpt that tells you what this post is about and is supposed to motivate you to click and find out more. ▸

What do you think leads to the article in the above layout? Well it's not obvious. But the only way to read more is to click on either the photo or that little red arrow right after the article lead. The entire content of the lead and the title are not clickable.

The easiest way to fix this is to remove the old linking and apply one anchor tag to the whole article. That way, nobody will have a problem to hit the tiny arrow and will quickly enter the article.


London
Sunny, sometimes overcast
27
10x
FASTER
London
Sunny, sometimes overcast
14°
27°
29 27 22
Tommorow Saturday Sunday

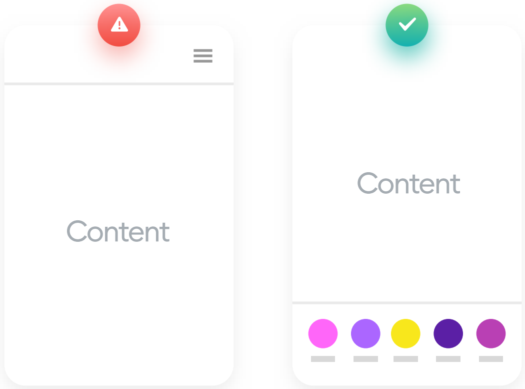
Albert Walicki

Title of the article

There is some post content here, like an excerpt that tells you what this post is about and is supposed to motivate you to click and find out more. ▸

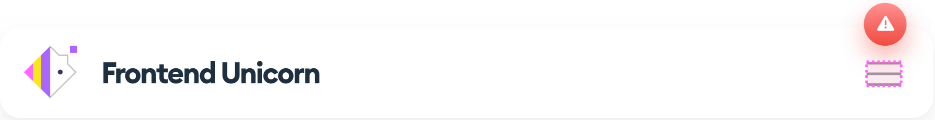
HAMBURGER MENU

Hamburger menus are not the best choice for mobile navigation. Hiding an entire navigation under just one button can be confusing and hard to udnerstand. Some users will simply miss it completely. The best scenario for mobile devices, is to go with either contextual navigation (links from within the content) or a tab bar at the bottom of the screen.

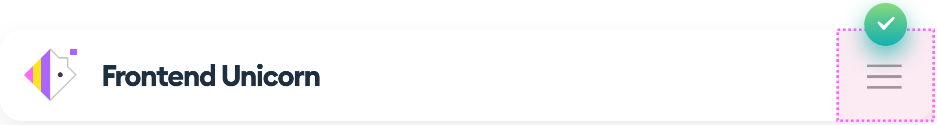


Whenever possible, try to use tabbed navigation, but remember to never exceed 5 tabs at the bottom of the screen. If you cannot fit the entire project onto 5 tabs, then the designer should likely re-think the navigation.

Sometimes, especially with complex, professional apps, hamburgers are a necessity. In such a case, they must have a larger clickable field than the end-of-dash bounding.



A bit of padding on the clickable element and everyone's life is better. Such menus can be tiny on phones. Adding padding to it won’t make it bigger visually, but the clickable area is now bigger and a lot easier to access.



“Hamburger menus are not the most accessible type of navigation. Whenever possible, go for visible tabs instead.”

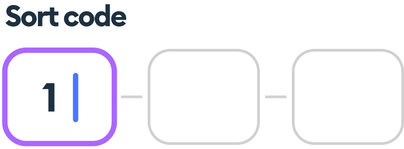
This point mainly concerns writing mobile and responsive web applications (tablet and phone). On desktop views, we have the option to jump to the next input using the TAB key. It's very convenient, and many of you are using it without even thinking about it.

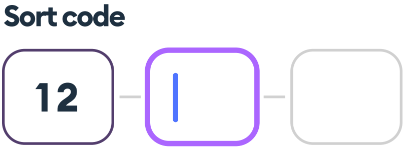
However, we do not have a TAB key on mobile devices.

It's something most companies, including the big ones, usually skip. Chances are, no one will tell you to do this on your project, but in my opinion, it is worth taking the time to implement.

Usually, it is best to implement this on enter click, but sometimes it is useful to auto-jump when the user enters the appropriate value.

Below we have an example from an application written in react native for one of our clients:



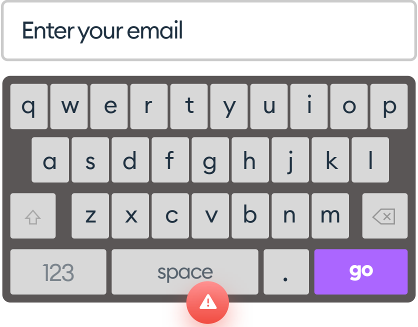


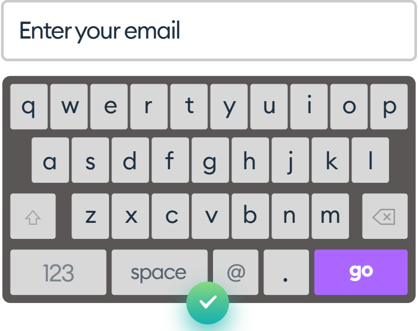
After the first two characters are typed, the focus jumps to the next field automatically. This is a strong acceleration of entering data by the user. Users do not have to click on two more inputs manually. Of course, if they make a mistake, they need to tap into the field again, but using this approach is still better, than manually going forward. You can also try allowing for the remove key on the mobile keyboard, to actually remove numbers between the fields as well.

With keyboards on mobile devices, there are two main things that most developers often overlook. Both of them make the lives of users a lot easier!

The first is to use the correct input types for the appropriate fields. How many times have you completed an email and the field was not of the email type? In this case the @ sign is usually on the second 'tab' of the keyboard.

We should adjust the keyboard, to the type of content each field needs. So in a case of an email field, we should end up with a mobile keyboard that incorporates the @ sign and the dot.

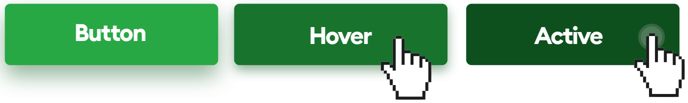

Enter your email
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
123
space
.
go


Enter your email
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
123
space
@
.
go

Doing this is super easy and quick, but the users get the feeling that someone has actually thought it through. And this translate directly to great user experience.

The second problem is the keyboard itself. Countless times I wanted to close keyboard after focus out, but it was still there. I wave the phone, click everything and finally somehow the keyboard manages to close itself.
It is really annoying but very easy to fix on the mobile app. Just remember about it.

The hover effect on links or buttons is an important part of the user experience. It informs the users, that they may interact with a given element by visually changing that element. An equally important element is to add 'cursor: pointer' to interactive objects, but it's a controversial topic among UX professionals. Some agree with this, and some strongly don't.

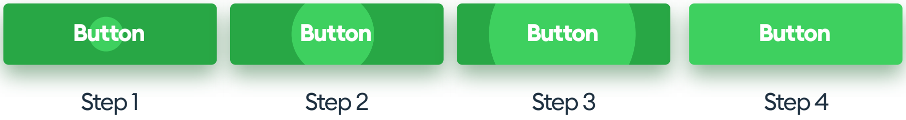


If the designer didn't provide you with a hover or active states in the project, it's best if you come up with them yourself. Luckily, we have a lot of effective examples of hover on buttons on the internet - 2d transitions, background, icons, border, shadow, and glow.

One of the most popular hover libraries is Hover.css written entirely in CSS. As part of learning, I recommend you try (without looking at the code) to copy a few effects from each category.

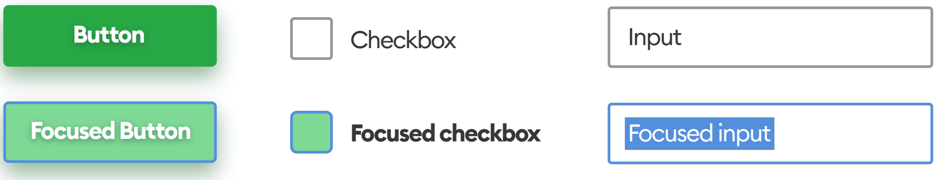
Of course, the hover effect only works on the desktop, as there is no way to "hover" over a button without touching it on a mobile device. Therefore, mobile users need different visual feedback; one of them may be the "Ripple effect". This effect shows the user that something has been pressed. One example is this CodePen by Ben Szabo.

Ripple effect animation example:



FOCUS STATE

Focus state helps the user find their way around the site, shows which input will be completed and which button is pressed. Not everyone uses a mouse to browse a website. Some users only use the keyboard, for various reasons, including certain disabilities. Focus also helps people with concentration problems and short-term memory loss - only because of clearly visible focus, they know where they are at any given moment.



If you can interact with an item through the mouse, you should be able to interact with the keyboard as well.

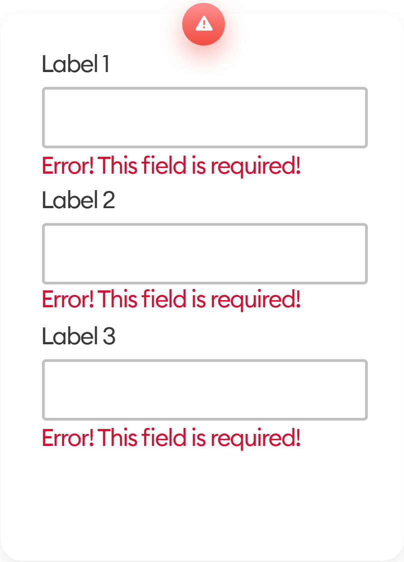
Badly implemented focus state is a very common accessibility issue, although it is very easy to fix. But what is the focus? Focus is something that "jumps" between the interactive elements on the page. Many different elements on the page can (and should!) be available to focus on them. All of them have a focus indicator to distinguish them from other non-focus elements.

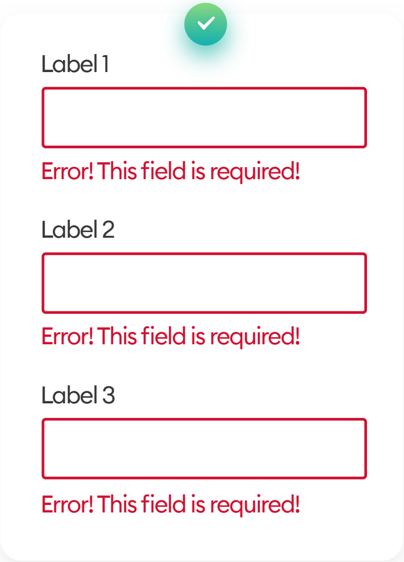
Here is a list of what you definitely need to make focusable:
- all links
- options and menus
- buttons
- inputs
- textareas
- selects
- custom pickers (calendar, time)

> “If you can interact with an item through the mouse, you should be able to interact with the keyboard as well.”

Error messages are often considered last when it comes to design and development. Many projects use some default variant of them without giving it a lot of thought. This approach deserves an error message of its own. There are a couple main issues with validation and error messages, that we believe you should know about.

But starting from the beginning, it's best to use an error message next to the field, instead of showing the errors near the bottom of the form. Always try to show the user the context of the error message, so they can quickly associate it with the field they need to fix.

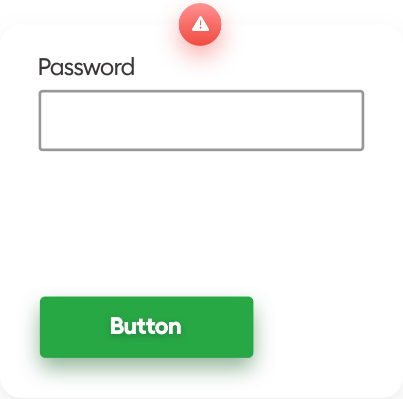

Label 1
Error! This field is required!
Label 2
Error! This field is required!
Label 3
Error! This field is required!

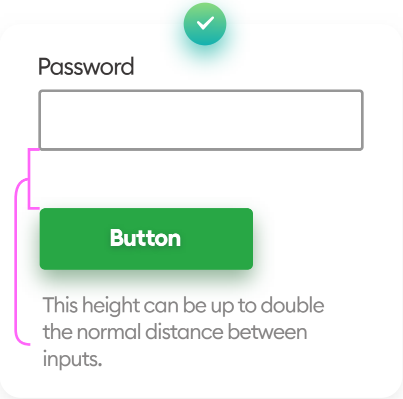

Label 1
Error! This field is required!
Label 2
Error! This field is required!
Label 3
Error! This field is required!

In the example on the left, the spacing between forms when the error messages are displayed makes all the fields blend into one, large column. The proper grouping of objects that are groups, is the most important part of estabilishing hierarchy and readability. In the second example, the spacing is bigger, so it's clearly visible which field (also with a red outline) you need to take care of.

If we are talking about the forms, it is impossible not to mention the buttons. The humble button is likely the most important UI element of all, because it leads to the main actions you perform on a website. If lets you "proceed", buy stuff or send messages.

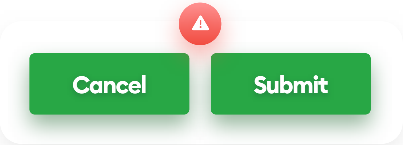
One potential problem you can face, is that the button is not visibly "connected" to the form. If the spacing between the last field and the button is too big, it may feel disconnected and mislead users that the button is not what actually sends the form. On a lighter note, you should also avoid vague button labels like "ok" or "next" and focus on the action the button takes you to instead. Good examples here can be "Submit", "Save changes" or "Create profile".

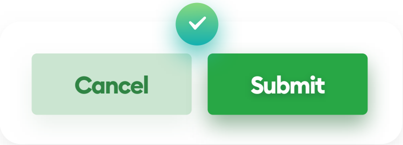




This height can be up to double the normal distance between inputs.

Another common problem happens when you have two buttons close to each other. You should always visually prioritize one of the buttons and they should NEVER look exactly the same. The more important call to action should be more prominent, while the secondary button can be just text, or a much lighter shade.





Knowing event pointers is useful for more complex layouts, where we have a large number of different elements overlapping each other. The pointer-event property specifies which event should respond to a particular HTML element.

Does that sound complicated? It's basically about:
- being able to click any HTML element (e.g. clicking a link or button)
- hovering the mouse over the HTML element area (hover effect)
- touching the HTML element on the screen without using the mouse (tap event)

Due to such possibilities, we can, for example, make an object inactive, so that we can interact with another object that lies underneath it.

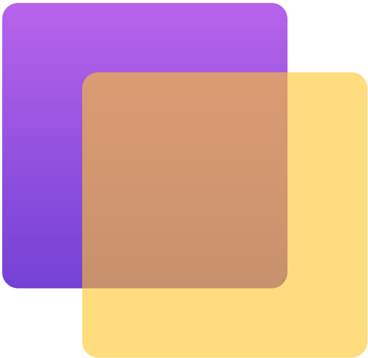
VALUES ASSUMED BY POINTER-EVENTS:	
none	prevents the object from being clicked, turns off all events such as hover
auto	is the default setting, enables all events if the parent is set to none, and the child - auto regains its default settings
inherit	inherits the properties of the parent element

Why is that helpful? It actually opens up quite a lot of possibilities for developers. How many times has it happened to you that you couldn't click on the arrows in a slider because another transparent object covered them?

With this property, without any physical layout changes, you can "disable the hidden object", and the arrows will work again.

Take **this JSFiddle snippet** as an example.

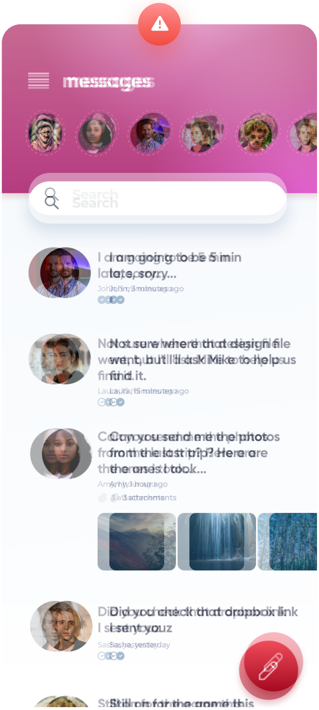
We have two squares there, one on top of the other. The square with the layer2 class is given pointer-events: none, which means that all mouse interactions are disabled.



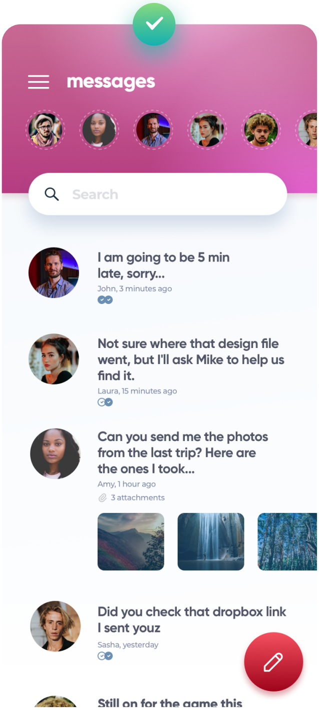
Due to this, hovering over this div starts hover for layer1! The second layer is completely invisible to the mouse.

If you comment the line 28 in the CSS area: "pointer-events: none;", then the second layer becomes "visible" to the mouse again, and the hover effect for the first layer fails.

One of the most useful front-end development tools is the pixel-perfect plugin. It allows you to overlay the actual design, at a reduced opacity, on top of your code preview. Obviously, some small changes (mostly in font thickness) are OK. But if it looks like an eye-exam gone wrong, you should move stuff around to fix it. It's the best way of working with a designer and being able to execute their vision exactly as they intended.



When you overlay that original screen on a coded design, you can clearly see that the objects are not in the right place.



messages

Search

I am going to be 5 min late, sorry...
John, 3 minutes ago

Not sure where that design file went, but I'll ask Mike to help us find it.
Laura, 15 minutes ago

Can you send me the photos from the last trip? Here are the ones I took...
Amy, 1 hour ago
3 attachments

Did you check that dropbox link I sent youz
Sasha, yesterday

Still on for the game this

Original app design screen.

If moving it by hand turns up very odd grid values (like each spacing being a completely different number) talk to your designer. Maybe the reference design wasn't yet fully completed on the grid part. Or maybe they used some other type of grid calculation. But it's good to understand where it came from.

Key takeaways

- ✓ hover and focus state are essential for a good User Experience
- ✓ there are multiple minor improvements that you can apply to your layouts. Those might be tabbing between inputs or using the correct input type
- ✓ always remember about the clickable area. Not all of your users have proficient computer skills. Make the active areas easy to access.

22.

Debugging

Debugging

CAN'T CODE WITHOUT BUGS

“If debugging is the process of removing software bugs, then programming must be the process of putting them in.”
— Edsger W. Dijkstra

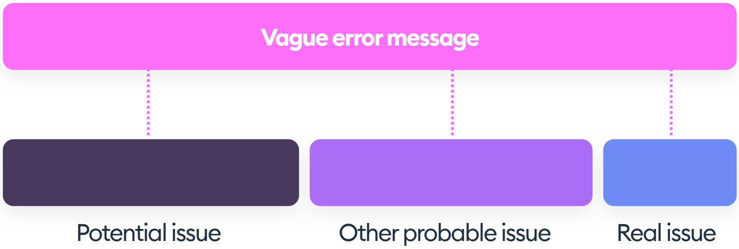
Edsger Dijkstra was often right, but that quote is one of my favorites. You can’t code without creating bugs. Some are relatively straightforward, like missing a semicolon or making a logical mistake in an if statement. Others are subtle and may happen only in specific datasets, browsers, or seemingly randomly.

Debugging is often dreaded by the programmers. Many of us got into programming to build things. Creating new code can be a satisfying endeavor; you can compare it to creating a bridge or skyscraper. Debugging is all about fixing things, more like plumbing or cleaning the shiny thing someone else had built. Definitely not fun.

Debugging is also challenging and tiring. You need to know your intended inputs and outputs and understand the app’s code and flow. Often finding a mistake is much harder than fixing it. But don’t fear, debugging can be learned and even mastered.

My main goal here is to teach you some tactics to find and fix errors. But I’d also like to convince you that you can find pride in debugging. It’s not shiny, but without maintenance, even the most beautiful constructions will fall. What’s more, debugging can teach you to discipline your thinking and open your eyes to the new horizons.

Why is debugging so demanding? Often the error we're getting is not directly connected to the cause. It may happen because of sloppy error messaging or complicated stack trace - sometimes the space between error and its symptom is vast.



Another obstacle to debugging is difficulty to reproduce an error. It's a typical situation when the end-user has some kind of a bug, but you can't see it on your machine. It may depend on configuration, different browser versions, or a different sequence of events. Sometimes bugs work in teams to mislead you. The bug may even subtly change during debugging.

This problem led to one of the most common developer jokes of "That's odd, it works on my end!", followed by uncomfortable laughter.

Also, one error can be caused by many various bugs. To make matters worse, often fixing a mistake can produce new ones. It usually indicates a lack of tests or sloppy fixes implemented without understanding the code.

Wow, that's quite a list. With so many things that can go wrong you should be proud of yourself when you find and fix a bug. But how to do that? Stick with me, we're getting there.

The naive approach to finding bugs is going through the code line by line, logging data, or setting breakpoints. You can use that in trivial apps or if you know almost exactly where the mistake is.

But typically, you need to debug a big application before you can familiarize yourself with the codebase. Making tons of logs will take forever. You have to do better than that. Here's a guide to finding any bug.

The first step is to find out what is working and what is not. To do that, you need to make sure that you understand the problem. The easiest way to limit your issue is to ignore most of the code. Separate code that works using return statements or comments. Try to remove as much of the code as possible and replicate the bug; the fewer code left, the easier it will be to debug it.

When you've limited the amount of code to analyze, it's time to reproduce the bug in different ways. Try other inputs or order of actions to check if the code works in different circumstances. Your goal is not to fix anything; you want to make sure you can produce the error every time. This way, you'll get the confidence that you understand what's going on.

The previous step may sound obsolete, but in fact, it's crucial. Many times you'll encounter errors that happen only for some users or cases.

For example, you may upload a file to the server, and everything works great. But when the user tries to do it, the upload fails. How to find the boundaries of that error? Ask the user (or check yourself):
- what is the file format,
- what is the file size,
- is the internet connection stable, and what happens when it's not.

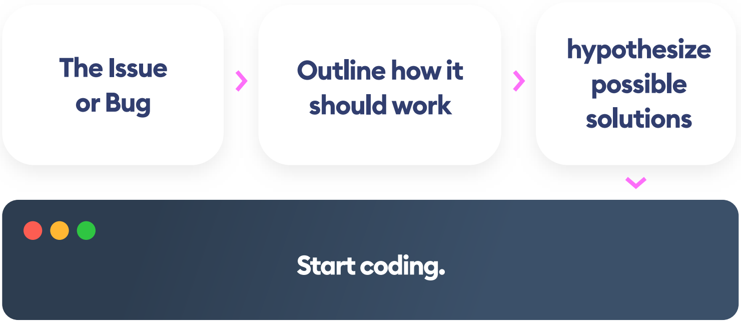
You'll see that minor details can make a difference between working code and error more often than not.

> “First step of debugging is to find out what IS working.”

Now you've got a reasonably comprehensive understanding of the problem so you can make some hypotheses.

Did you notice that we didn't start to code yet?

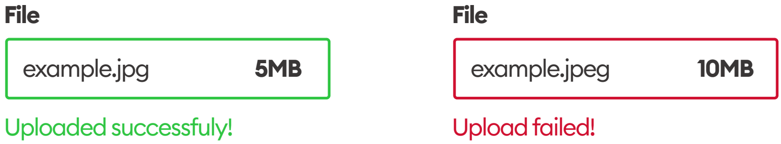
That's correct; debugging is about creating a precise mental model of how the code works and what's wrong with it. With your knowledge about cases that work and cases that don't, you should be able to produce some possible solutions.



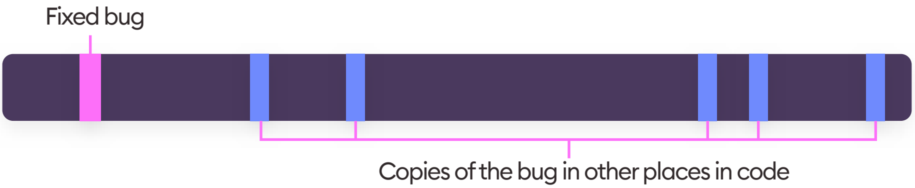
Only now we're ready to do some coding. Try your hypothesis, starting from the most probable one. But stick to one possible solution at a time!

In the next section, I'll show you some useful techniques for testing hypotheses, but for now, let's get back to the file upload example. So, you upload a 1MB PDF file, and everything works. The client uploads 5MB PNG, and the upload fails. Our first hypothesis is that the server doesn't permit PNGs to be uploaded. So we can start by uploading 1MB (not 5MB) PNG. If the upload fails, we're lucky and can start fixing the problem. If we're not, we can go to the second hypothesis - we can't upload bigger files. So try uploading 5MB PDF (not PNG). If the upload fails, we've got a probable culprit; if not, we need to search deeper. Maybe there is a CORS error or the client has slow connection and the upload times out?

Let’s assume that one of the solutions above worked, and we fixed the problem. Are we done? Not really. Now we need to think if there are possible other bugs related to this same problem. Maybe now we can upload a 5MB file, but not 10MB? What is the maximum size of the file we should accept?



Or perhaps we can’t upload JPGs? How about JPEGs? You need to check different combinations of possible problems and make sure your solution meets the clients requirements. Often, this same error can be produced in different ways, and the issue can have more than one cause.

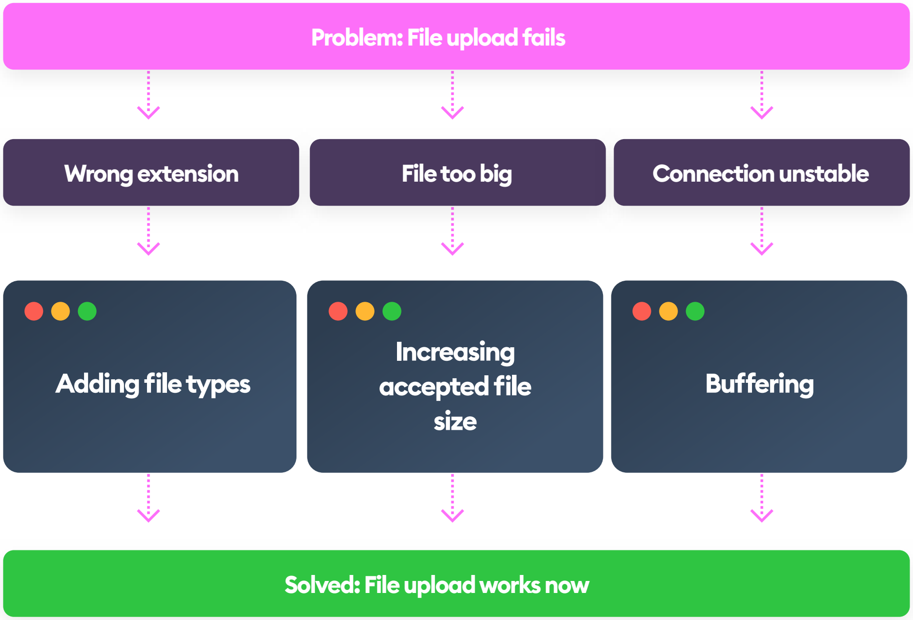


Also, make sure that an identical bug doesn’t exist in other parts of the codebase. I hope you followed the DRY (don’t repeat yourself) principle and didn’t copy and paste code, but maybe other, slightly different use cases need to be fixed?

When you fixed the bug, check the first failed case and all other failed scenarios that you created. It’s also good to make some tests to prevent this bug from happening in the future. Depending on your company policy, it may not be possible, as many companies don’t do testing. In that case, try at least to document a bug and your solution in the comments.

Okay, you’re done! You fixed the bug.

I've outlined the process of debugging; now is an excellent time to give you more specific tools to find bugs in your code. These tools are based on a divide and conquer technique.


Problem: File upload fails
Wrong extension
File too big
Connection unstable
Adding file types
Increasing accepted file size
Buffering
Solved: File upload works now

Divide and conquer is an algorithm design type. It works by recursively breaking a problem into two (or more) smaller issues of a similar nature. If you can solve the minor issue, you can solve a bigger one too. Divide and conquer is used primarily for sorting or finding the closest pair of points.

But divide and conquer is more than that. It's a mindset you can use to debug your apps.

> “Divide and conquer works by recursively breaking a problem into two (or more) smaller issues of a similar nature.”

The straightforward use of divide and conquer in debugging is to split the codebase in half and add a log there. If the log works and prints correct data, we can assume that everything up to that point works fine; if not, the error is in the first half of the code.

Next, we take the suspicious half of the code and put another log in the middle. We repeat the procedure until we discover the issue. Each time we cut the remaining codebase in half - so we should find the error in just a few steps. That's a good technique, a lot better than logs all over the code. But we can go a step further.

Educated Guesses and Divide and Conquer
Imagine your app consists of thousands of lines. It's hard to find the middle, and you'll need dozens of logs to discover the culprit. Now it's the time to take a step back and think before adding logs.

Start with answering a few questions and making some hypotheses. What broke? Is it likely that some string transformation failed; maybe the data returned is not in the type you expected, or is there another issue?

Do you have some tests or checked parts of the code? If so, the bug probably isn't there. When did the bug occur? Did everything work fine before? What changed? Maybe inputs are different now, or perhaps the code worked two commits back.

When you answer these questions, you're ready to make an educated guess about the bug's nature. Every assumption can prove wrong, but your goal isn't to be correct. It's to find the most probable culprit so you can pinpoint the suspicious parts of code.

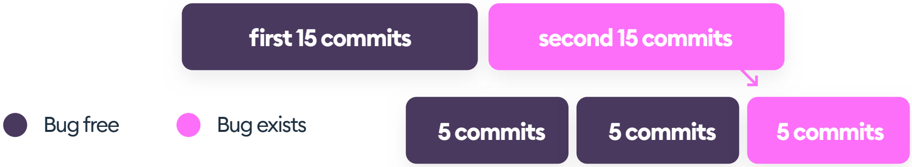
When you have a few hypotheses, zoom in on the most likely and use divide and conquer. If you find the bug, that's great. If not, proceed to the next hypothesis.

Imagine working on an application for a few years. You've just made some minor modifications. All of a sudden, you got a new bug. Everything worked fine before, and you've changed just a few code lines, so you naturally check them to see what's broken. But everything in the new code works fine. How is that possible? And where to look for an error in a vast codebase?

These kinds of bugs are called regression bugs - they're unexpected errors unrelated to the code's recent changes. What to do now? First, you need to use previously described techniques to find part of the code that's probably wrong.

When you've got that, you should "get back in time" to the moment when the bug wasn't there. You can do that using the command git checkout <COMMIT_SHA> (you can check SHA of the commit in any git tool you're using). That will let you go to the state of the code at the time of that commit.

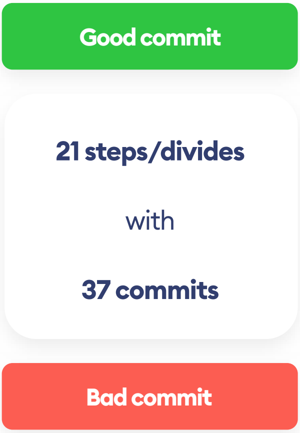
So let's say you've checked out to the commit from the week before. Check if the bug is already there. If it's not, something was broken between now and then. If the code from last week is buggy, you have to look further back. When you finally find the commit where the code worked, you have to find the exact commit that broke the feature. Let's say that code worked a week ago, and it's not working now; there are 30 commits in-between. How to find the wrong one? Use divide and conquer!



Split the number of commits in half; in the case above, you need to skip the first 15 commits, and you'll know if the error is in the first or the second fifteen. Repeat the process until you'll find a culprit.

That approach to finding bugs in time is so common that git has a special command that helps you do that faster. Firstly, check out to the latest commit; for example, if you're on the master branch, use git checkout master.

Then start bisecting by git bisect start. We need to indicate to GIT the first and last commit of the bisect. Our latest commit has the bug, so we mark it as broken with git bisect bad HEAD. Then we mark the commit when the code worked fine with git bisect good <COMMIT_SHA>.



Git will tell you how many commits are between the good and the bad commit and how many steps (divides) are needed to find the commit that broke something.

The bisect will check you out to the commit in the middle between the good and the bad. Now you need to test the code. If it's okay, use the command git bisect good if not, use git bisect bad. Repeat the process until you find the culprit, and then run git bisect reset to get back to the HEAD commit and fix the error.

Debugging can be a stressful and tiring process. You have to approach it with calm and confidence.

Beware of the simplest debugging mistake - if something works for you and doesn’t for other people, make sure you’re on the right branch. I’ve been on the wrong branch more times than I’d like to admit.

If you get stuck and have no idea what to do next, talk with another developer. You may not need their help, but the process of explaining the error and your attempts to fix it can lead to the discovery of the bug. Talking with others lets you organize your thinking better, and an additional pair of eyes is always useful.

Key takeaways

- ✓ debugging is done mainly in the head
- ✓ always try to narrow down the problem as much as possible
- ✓ debugging is a huge part of programming - better get used to it
- ✓ it's usually easier to fix a bug than find it
- ✓ divide and conquer is your best friend in debugging

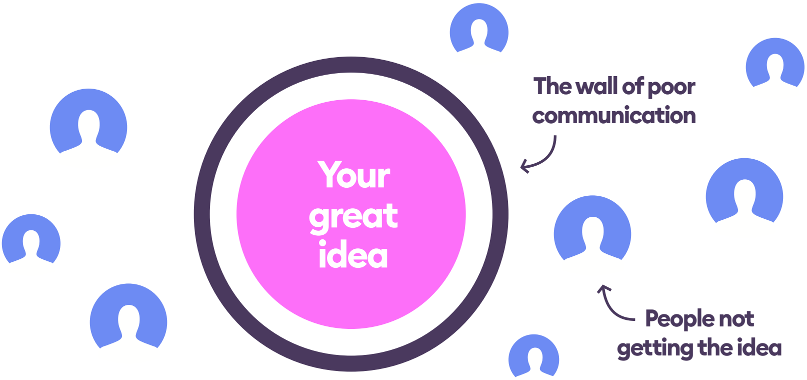
23.

Communication skills

Communication skills

COMMUNICATE!

Programmers are mainly communicators. You communicate with other people verbally, and non-verbally. And you are communicating with them with your code. Master those two aspects of communication, and you're bound to be successful.


Your great idea
The wall of poor communication
People not getting the idea

Social skills
Teams create great things. It doesn't matter how good a programmer is; you can't do everything by yourself. Serious programming is a team sport, not something done in isolation. Even if you think you're working alone, you're in a space full of other people - be it other developers, managers, or clients.

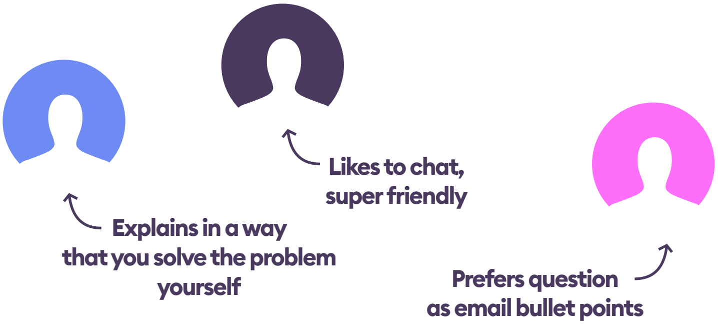
The ability to communicate with others is a massive advantage. Efficient, precise, and polite communication is among developers' essential skills, especially for developers aspiring to senior positions.

To be clear - you don't have to be friends with the people you work with. All you need to do is be a good team member - calm, empathetic, and attentive.

You may believe communication with fellow developers is most straightforward. After all, you use similar jargon, may share attitudes, priorities, and interests. But it's trickier than you may imagine.

I once worked with a senior developer who frequently seemed to ignore my questions and remarks. He just kept on coding as if I wasn't even there. I would stand there disoriented, unsure if I should repeat the question, wait, or just leave him be. After a minute or two (or five), he would respond, always on the subject, and provide useful guidance.

It turned out he heard me. But he preferred to answer after he finished the task at hand. He wasn't unfriendly or unhelpful; he just needed to take his time to respond. I could get angry waiting, but fortunately, I realized how to work with him.



People differ from each other, and you need to take it into account. Devote some time to get to know people you work with, their habits, and their communication styles. You'll work a lot faster if you know who to ask for help and how to do it.

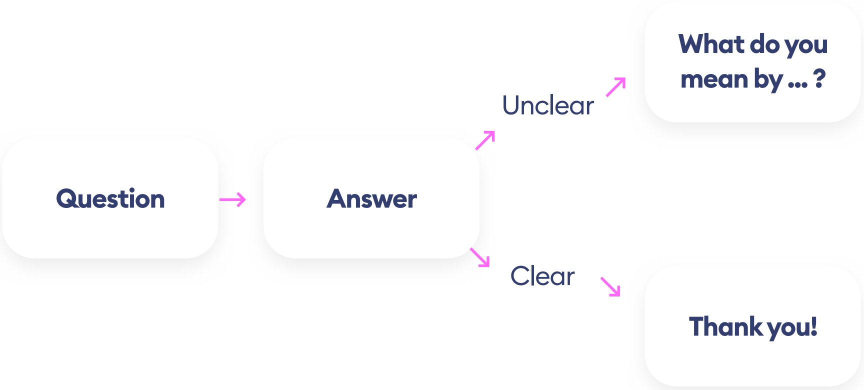
To avoid unproductive conflicts, follow a few simple rules. Treat everyone with respect, no matter what their position is. It doesn't cost you anything but improves the working environment.

“

Treat everyone with respect, no matter what their position is ”

Try to be a good listener. That's an important and often neglected skill. Train yourself in listening to understand, not just to respond. Assume everyone you work with has something to say that's useful for you. Aspire to be attentive in your listening, interpret things clearly, and ask for clarification if something seems convoluted or unclear.

Don't be afraid to ask, if the answer is unclear. Some people fear, that not understanding something is their fault. It's not. Ask as much as you need to understand the answer! A conversation should leave both you and your counterpart better of.



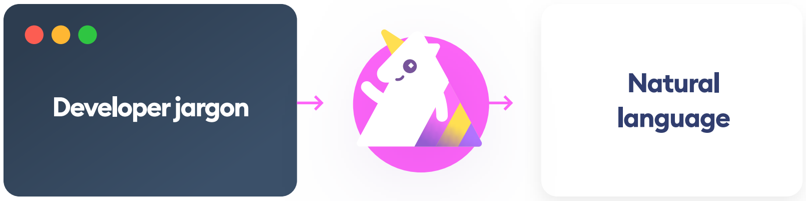
Soft skills are also crucial when you're receiving or giving feedback. It doesn't matter if you're a mentor or trainee – you have to treat feedback as a learning opportunity. Being polite and considerate is a must.

You'll have more friends if you don't take feedback too personally and give it with empathy. The single best advice about giving feedback is to deliver your doubts as questions. Don't order others to do things; ask them to consider other approaches. Try it, and you'll see how much it changes the conversation dynamic.

If you can't get along with managers and clients, you won't be a great developer.

Essentials of communication with them are similar to communication with developers. Being pleasant, calm, and empathetic will get you far. The biggest mistake you can make is to assume your work is somehow more important than theirs. You're all working as a team and need to treat each other with respect.

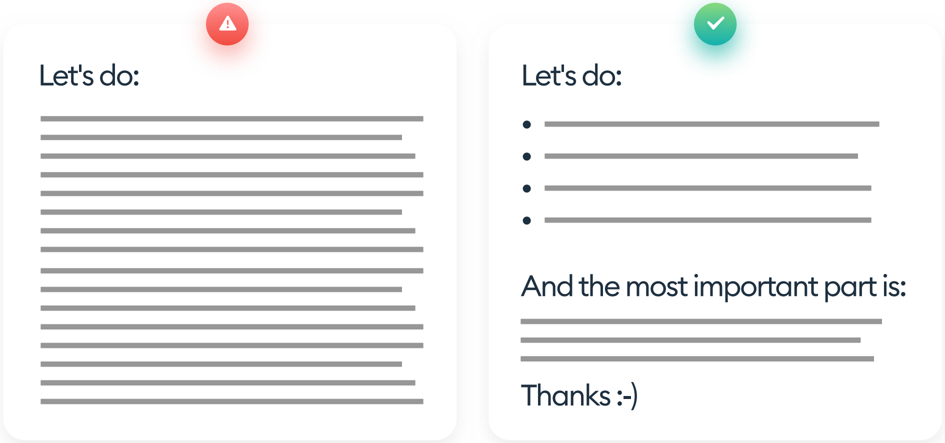
Programmers have their language and so do the business people. You have to remember that not everyone has your expertise, and often you need to explain complex topics in simple terms.



Developers who can connect business and programming departments are crucial for the product's success. By acting as a bridge, you can make more friends and accelerate your team's work. To do that, you need to consciously try to learn the language of other experts in your team. Treat it like learning a new coding language, and then master it.

Countless potentially significant projects failed due to miscommunication. Don't let it happen to yours – experience in another business may be critical in developing your company and your programming career.

A considerable part of your communication will be in writing. To be an effective communicator in writing, you should follow two rules:
make it short, make it pleasant.



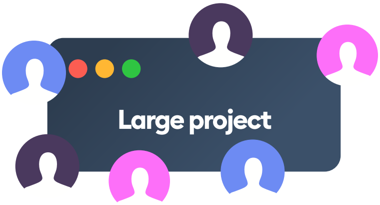
Communicate your ideas in writing using short sentences and paragraphs. Create an easy to understand structure and skip all the unnecessary words. The ideal message is concise, simple, and straightforward. Most of the time, you should limit yourself to 3-5 sections. Sometimes even two or three sentences are enough. By limiting yourself, you're more likely to communicate only the main points and do it clearly, but also you respect the time of all the people who're to read your message.

The second crucial thing is being nice. Written words can have a massive impact on others and contribute to conflicts. Usually, people seem more coarse and unpleasant in writing. So take your time to read what you wrote one more time, choose your words carefully, and be kind. If it's appropriate in your organization use positive emojis to enhance good vibes.

Many young programmers think about coding as an interaction between them and a machine. That's only a partial truth. Of course, the computer needs to understand your instructions to execute them. But in reality, programmers need to communicate not only with the machine but with other programmers too. As programming legend put it:

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
Martin Fowler.

Almost every non-trivial application is created by a team. What's more, code is read much more often than it is written. You need to communicate with other programmers efficiently, so making the code readable is crucial. It helps to work faster, simplifies maintenance, and facilitates further development.



You may think your side project's readability is not essential. After all, you're the only developer, and you know what you're doing. But do you? Imagine you need to take a break from the project for half a year. Are you sure you'll understand that code line or that magical number in six months? I don't think so.

Think of your code as a map. A map only you can use is, well, useless. Your map is for other people. Everyone who understands the map-reading basics should get it. Strive for this level of clarity in your code.

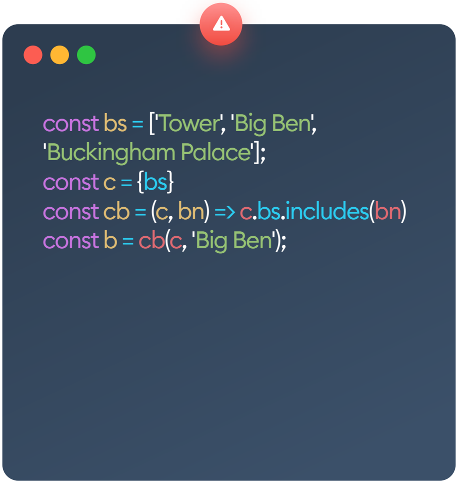
On the following pages I'll share some tips to make your code more like a map and less like a personal diary.

> “Code is read more often than it is written.”

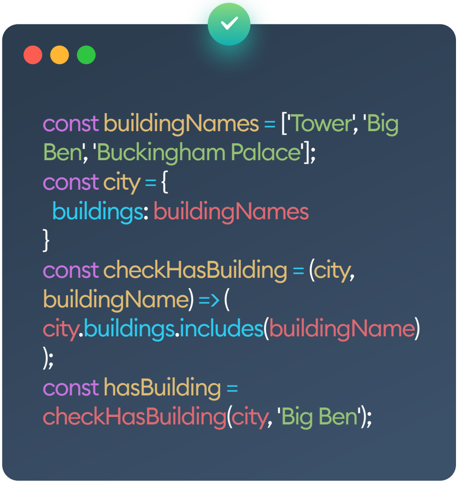
You can implement a tree traversal algorithm in 80 characters. That’s impressive, but what about all the programmers who don’t have your skill?

Next month or next year, someone will bump into your code and need to add or change something. How long will it take to understand the codebase? Writing code is not a contest. You are a part of the team.

Your team doesn’t want to be impressed by you – just to understand. When we want to be astonished, we can take a look at your code katas. In our project, we’d instead appreciate your consideration.



```
const bs = ['Tower', 'Big Ben',
'Buckingham Palace'];
const c = {bs}
const cb = (c, bn) => c.bs.includes(bn)
const b = cb(c, 'Big Ben');
```



```
const buildingNames = ['Tower', 'Big
Ben', 'Buckingham Palace'];
const city = {
  buildings: buildingNames
}
const checkHasBuilding = (city,
buildingName) => (
city.buildings.includes(buildingName)
);
const hasBuilding =
checkHasBuilding(city, 'Big Ben');
```

The worst form of trying to impress is using short, meaningless variable, and function names. Your primary goal is to create code as readable as possible. The golden standard is the code that can be followed by someone who is not a programmer, only reads in plain English. It’s worth being slightly more verbose if it means the code is easier to understand for others.

To become a successful programmer, you need to follow ground rules. Countless problems have already been solved. You should take advantage of it.

Follow the patterns created by your predecessors, and you'll be understood. You don't need to be a slave to others' opinions. Just be sure you know why you're breaking convention and if it's worth it.

You can find countless books about design patterns and software conventions. It's not the aim of this book to describe them; many phenomenal books on the subject are already on the market. But there is a single principle that tends to be crucial for writing readable code:

the single-responsibility principle.

This principle's essence is that every function (or method) should be small and do only one thing, and do it well. Junior developers often forget about it and write big functions that handle multiple functionalities. It's better to write many smaller functions, name them reasonably, and compose bigger functionality out of them.

One easy rule to help you write small functions is to make sure they don't take too many arguments. Usually, if you need more than three arguments, the function tries to do many things. When possible, try to stick to only one argument.

You should follow not only the conventions in code design but also in formatting. You have to use tools that automate your formatting, like Prettier. It will improve consistency, which enhances readability.

REMOVE UNNECESSARY CODE

The longer it takes to create the application, the more cluttered the code will be. Remember to get rid of unnecessary things impeding code understanding.

Remove unused variables, misleading comments, and overly complicated abstractions. Leave only the code that solves the problem you need to address.

Also, make the code more concise as long as it doesn't impair the readability.

Pay special attention to removing code when you're adding or removing features. Codebase tends to grow every month, and over time it's easy to get overwhelmed with unused code. So try to remove something every time you do some more significant changes.

Let the scout's rule guide your fixes:
leave your code better than you found it!

Key takeaways

✓ being polite and considerate will get you far

✓ good written communication is short communication

✓ social skills are essential - a huge part of your work in cooperation with other people

✓ emails always sound harsher for your reader than for you - make a special effort to make them nice

✓ you communicate with others also by code - its readability should be your major priority

24.

Useful tools

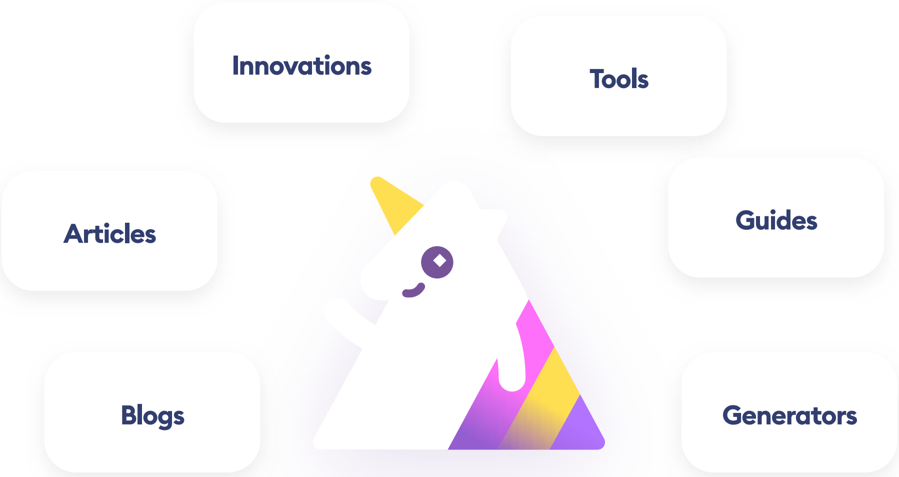
Useful tools

RECOMMENDED RESOURCES

Even the greatest programmers won't get far by themselves. To be productive, you have to learn to leverage the work of others.

We prepared a curated list of tools and resources for front-end developers to help you with this task. Some of the links below provide detailed knowledge about specific subjects, like CSS or JavaScript. Others are the tools you'll use daily to improve your developer experience - they can help with all kinds of things, from SEO to getting placeholder images. We plan to add to this list with our future updates!

Have fun exploring the list, and good luck with your programming journey!



MAIN SOURCES OF KNOWLEDGE

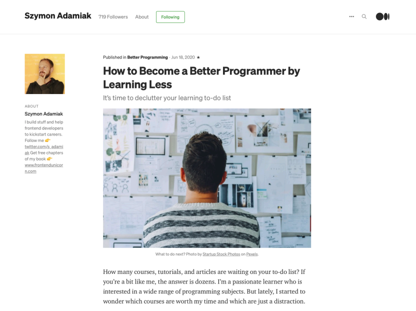
https://stackoverflow.com/

https://medium.com/

https://css-tricks.com/

https://github.com/

OUR MEDIUM BLOGS



ABOUT

Szymon Adamiak

I build stuff and help frontend developers to kickstart careers. Follow me 👉 twitter.com/s_adamiak Get free chapters of my book 👉 www.frontendunicorn.com

Published in **Better Programming** · Jun 18, 2020 ★

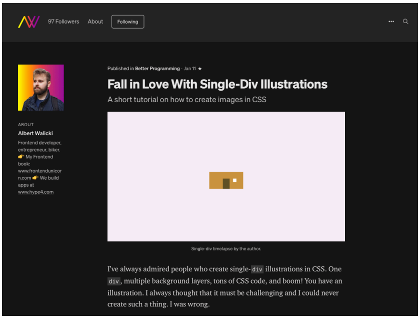
How to Become a Better Programmer by Learning Less

It's time to declutter your learning to-do list

What to do next? Photo by Startup Stock Photos on Pexels.

How many courses, tutorials, and articles are waiting on your to-do list? If you're a bit like me, the answer is dozens. I'm a passionate learner who is interested in a wide range of programming subjects. But lately, I started to wonder which courses are worth my time and which are just a distraction.

https://medium.com/@sadamiak



ABOUT

Albert Walicki

Frontend developer, entrepreneur, biker. 👉 My Frontend book: www.frontendunicorn.com 👉 We build apps at www.hype4.com

Published in **Better Programming** · Jan 11 ★

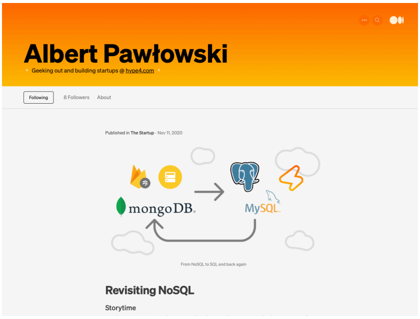
Fall in Love With Single-Div Illustrations

A short tutorial on how to create images in CSS

Single-div timelapse by the author.

I've always admired people who create single-`div` illustrations in CSS. One `div`, multiple background layers, tons of CSS code, and boom! You have an illustration. I always thought that it must be challenging and I could never create such a thing. I was wrong.

https://albertwalicki.medium.com


Albert Pawłowski

⚡ Geeking out and building startups @ hype4.com ⚡

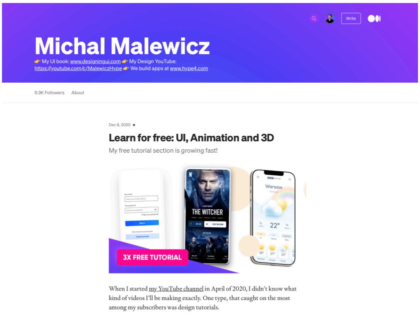
Published in **The Startup** · Nov 11, 2020

From NoSQL to SQL and back again

Revisiting NoSQL

Storytime

https://albertpawlowski.medium.com


Michal Malewicz

👉 My UI book: www.designingui.com 👉 My Design YouTube: https://youtube.com/c/MalewiczHype 👉 We build apps at www.hype4.com

Dec 8, 2020 ★

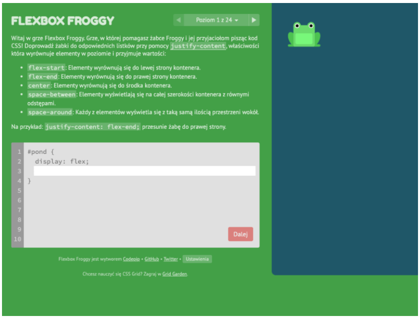
Learn for free: UI, Animation and 3D

My free tutorial section is growing fast!

When I started my YouTube channel in April of 2020, I didn't know what kind of videos I'll be making exactly. One type, that caught on the most among my subscribers was design tutorials.

https://michalmalewicz.medium.com

EDUCATION GAMES



Witaj w grze Flexbox Froggy. Grze, w której pomagasz żabce Froggy i jej przyjaciołom pisząc kod CSS! Doprowadź żabki do odpowiednich listków przy pomocy `justify-content`, właściwości która wyrównuje elementy w poziomie i przyjmuje wartości:

- `flex-start`: Elementy wyrównują się do lewej strony kontenera.
- `flex-end`: Elementy wyrównują się do prawej strony kontenera.
- `center`: Elementy wyrównują się do środka kontenera.
- `space-between`: Elementy wyświetlają się na całej szerokości kontenera z równymi odstępami.
- `space-around`: Każdy z elementów wyświetla się z taką samą ilością przestrzeni wokół.

Na przykład: `justify-content: flex-end;` przesunie żabę do prawej strony.

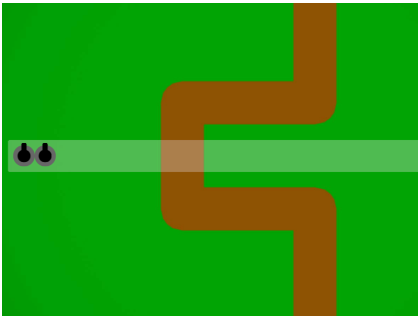
```
#pond {
  display: flex;

}
```

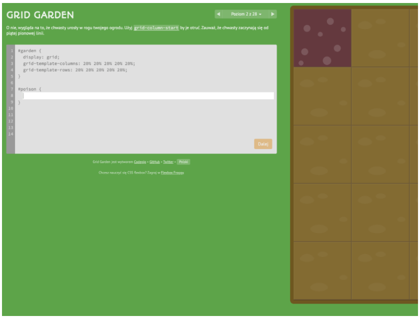
Flexbox Froggy jest wytworem Codepip • GitHub • Twitter •

Chcesz nauczyć się CSS Grid? Zagraj w Grid Garden.

https://flexboxfroggy.com/



http://www.flexboxdefense.com/



O nie, wygląda na to, że chwasty urosły w rogu twojego ogrodu. Użyj `grid-column-start` by je otruć. Zauważ, że chwasty zaczynają się od piątej pionowej linii.

```
#garden {
  display: grid;
  grid-template-columns: 20% 20% 20% 20% 20%;
  grid-template-rows: 20% 20% 20% 20% 20%;
}

#poison {

}
```

Grid Garden jest wytworem Codepip • GitHub • Twitter •

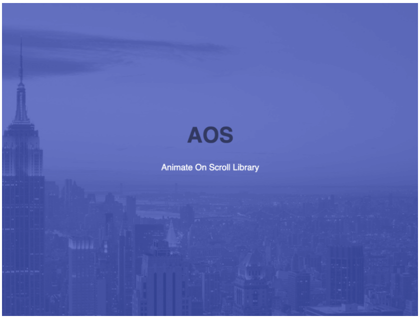
Chcesz nauczyć się CSS flexbox? Zagraj w Flexbox Froggy.

https://cssgridgarden.com/

Animate.css

Just-add-water CSS animations

https://animate.style/

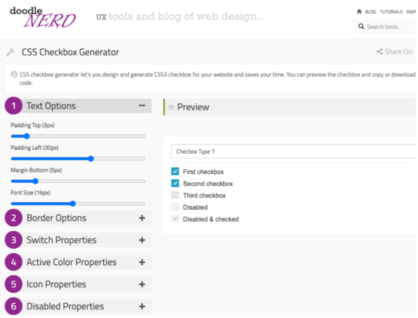


AOS

Animate On Scroll Library

https://michalsnik.github.io/aos/

GENERATORS


doodle NERD
UX tools and blog of web design..
BLOG
TUTORIALS
SNIP
Search tools..
Share On:
1 Text Options
2 Border Options
3 Switch Properties
4 Active Color Properties
5 Icon Properties
6 Disabled Properties
Preview

CSS Checkbox Generator

CSS checkbox generator let's you design and generate CSS3 checkbox for your website and saves your time. You can preview the checkbox and copy or download code.

Padding Top (3px)

Padding Left (30px)

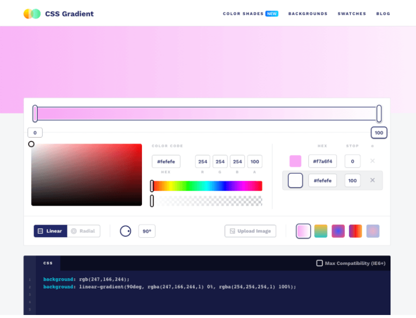
Margin Bottom (5px)

Font Size (16px)

Checbox Type 1

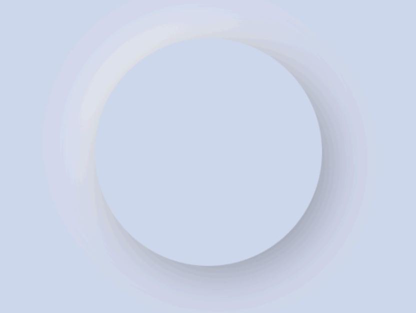
- ☑ First checkbox
- ☑ Second checkbox
- ☐ Third checkbox
- ☐ Disabled
- ☑ Disabled & checked

http://doodlenerd.com/

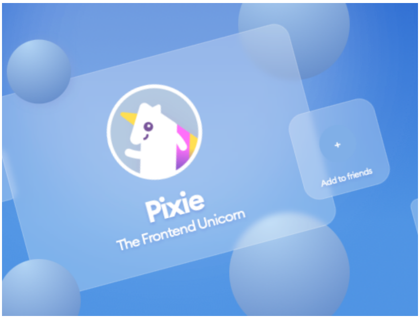

CSS Gradient
COLOR SHADES NEW
BACKGROUNDS
SWATCHES
BLOG
0
100
COLOR CODE
#fefefe
254
254
254
100
HEX
STOP
#f7a6f4
0
#fefefe
100
Linear
Radial
90°
Upload Image
css
☐ Max Compatibility (IE6+)

```css
background: rgb(247,166,244);
background: linear-gradient(90deg, rgba(247,166,244,1) 0%, rgba(254,254,254,1) 100%);
```

https://cssgradient.io/

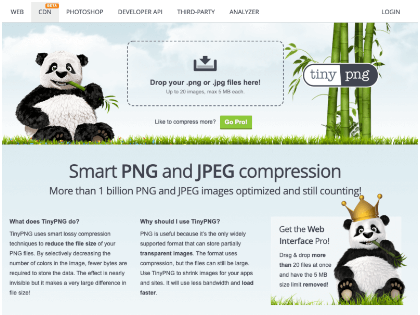


https://neumorphism.com


+
Add to friends
Pixie
The Frontend Unicorn

https://glassmorphism.com

IMAGE OPTIMIZATION


WEB
CDN
BETA
PHOTOSHOP
DEVELOPER API
THIRD-PARTY
ANALYZER
LOGIN
tiny png
Go Pro!

Drop your .png or .jpg files here!

Up to 20 images, max 5 MB each.

Like to compress more?

Smart PNG and JPEG compression

More than 1 billion PNG and JPEG images optimized and still counting!

What does TinyPNG do?

TinyPNG uses smart lossy compression techniques to **reduce the file size** of your PNG files. By selectively decreasing the number of colors in the image, fewer bytes are required to store the data. The effect is nearly invisible but it makes a very large difference in file size!

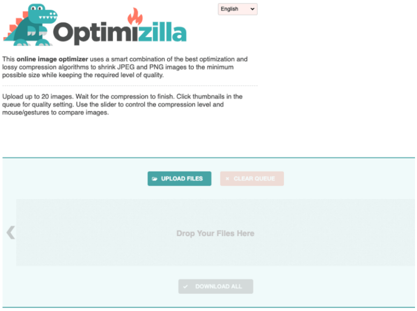
Why should I use TinyPNG?

PNG is useful because it's the only widely supported format that can store partially **transparent images**. The format uses compression, but the files can still be large. Use TinyPNG to shrink images for your apps and sites. It will use less bandwidth and **load faster**.

Get the **Web Interface** Pro!

Drag & drop **more than** 20 files at once and have the 5 MB size limit **removed!**

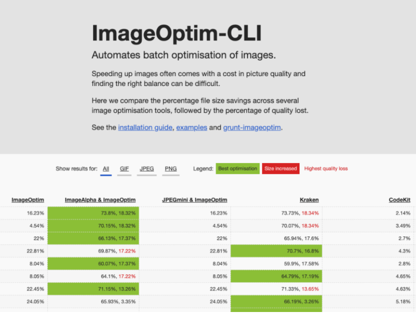
https://tinypng.com/


Optimizilla
English
UPLOAD FILES
CLEAR QUEUE
Drop Your Files Here
DOWNLOAD ALL

This **online image optimizer** uses a smart combination of the best optimization and lossy compression algorithms to shrink JPEG and PNG images to the minimum possible size while keeping the required level of quality.

Upload up to 20 images. Wait for the compression to finish. Click thumbnails in the queue for quality setting. Use the slider to control the compression level and mouse/gestures to compare images.

https://imagecompressor.com/


Show results for: All GIF JPEG PNG
Legend: Best optimisation Size increased Highest quality loss

ImageOptim-CLI

Automates batch optimisation of images.

Speeding up images often comes with a cost in picture quality and finding the right balance can be difficult.

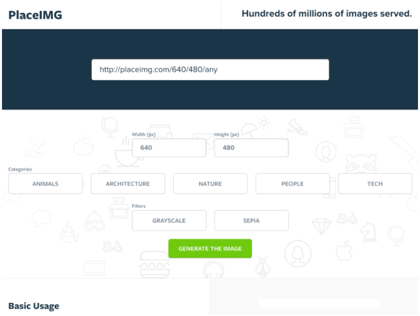
Here we compare the percentage file size savings across several image optimisation tools, followed by the percentage of quality lost.

See the installation guide, examples and grunt-imageoptim.

ImageOptim	ImageAlpha & ImageOptim	JPEGmini & ImageOptim	Kraken	CodeKit
16.23%	73.8%, 18.32%	16.23%	73.73%, 18.34%	2.14%
4.54%	70.15%, 18.32%	4.54%	70.07%, 18.34%	3.49%
22%	66.13%, 17.37%	22%	65.94%, 17.6%	2.7%
22.81%	69.87%, 17.22%	22.81%	70.7%, 16.8%	4.3%
8.04%	60.07%, 17.37%	8.04%	59.9%, 17.58%	2.8%
8.05%	64.1%, 17.22%	8.05%	64.79%, 17.19%	4.65%
22.45%	71.15%, 13.26%	22.45%	71.33%, 13.65%	4.63%
24.05%	65.93%, 3.35%	24.05%	66.19%, 3.26%	5.18%

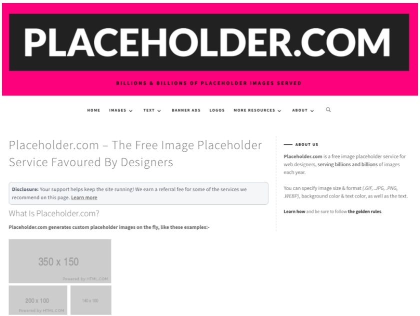
http://jamiemason.github.io/ImageOptim-CLI/

PLACEHOLDER IMAGES


PlaceIMG
Hundreds of millions of images served.
Width (px)
640
Height (px)
480
Categories
ANIMALS
ARCHITECTURE
NATURE
PEOPLE
TECH
Filters
GRAYSCALE
SEPIA
GENERATE THE IMAGE
Basic Usage

http://placeimg.com/640/480/any

http://placeimg.com/


PLACEHOLDER.COM
BILLIONS & BILLIONS OF PLACEHOLDER IMAGES SERVED
HOME
IMAGES
TEXT
BANNER ADS
LOGOS
MORE RESOURCES
ABOUT
350 x 150
Powered by HTML.COM
200 x 100
Powered by HTML.COM
140 x 100

Placeholder.com – The Free Image Placeholder Service Favoured By Designers

Disclosure: Your support helps keep the site running! We earn a referral fee for some of the services we recommend on this page. Learn more

What Is Placeholder.com?

Placeholder.com generates custom placeholder images on the fly, like these examples:-

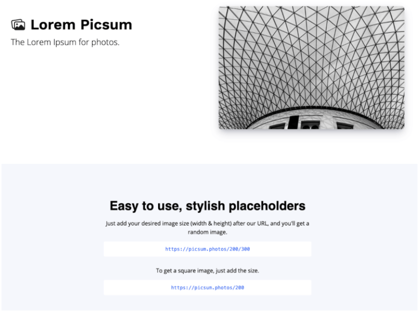
ABOUT US

Placeholder.com is a free image placeholder service for web designers, **serving billions and billions** of images each year.

You can specify image size & format *(.GIF, .JPG, .PNG, .WEBP)*, background color & text color, as well as the text.

Learn how and be sure to follow **the golden rules**.

https://placeholder.com/


Lorem Picsum

The Lorem Ipsum for photos.

Easy to use, stylish placeholders

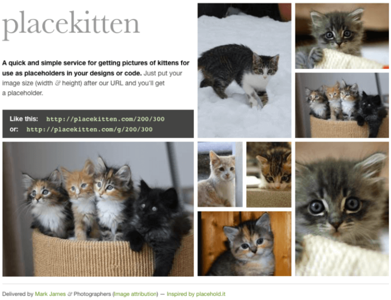
Just add your desired image size (width & height) after our URL, and you'll get a random image.

`https://picsum.photos/200/300`

To get a square image, just add the size.

`https://picsum.photos/200`

https://picsum.photos/

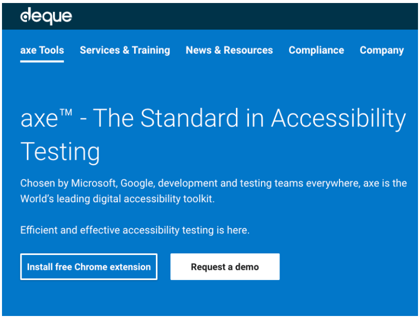

placekitten

A quick and simple service for getting pictures of kittens for use as placeholders in your designs or code. Just put your image size (width & height) after our URL and you'll get a placeholder.

Like this: `http://placekitten.com/200/300`
or: `http://placekitten.com/g/200/300`

Delivered by Mark James & Photographers (Image attribution) — Inspired by placehold.it

https://placekitten.com/

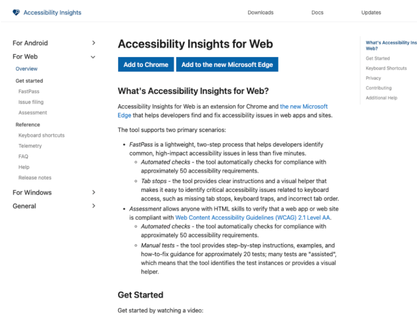

deque
axe Tools
Services & Training
News & Resources
Compliance
Company
Install free Chrome extension
Request a demo

axe™ - The Standard in Accessibility Testing

Chosen by Microsoft, Google, development and testing teams everywhere, axe is the World's leading digital accessibility toolkit.

Efficient and effective accessibility testing is here.

https://www.deque.com/axe/


Accessibility Insights
Downloads
Docs
Updates
For Android
For Web
Overview
Get started
FastPass
Issue filing
Assessment
Reference
Keyboard shortcuts
Telemetry
FAQ
Help
Release notes
For Windows
General
Add to Chrome
Add to the new Microsoft Edge
What's Accessibility Ins Web?
Get Started
Keyboard Shortcuts
Privacy
Contributing
Additional Help

Accessibility Insights for Web

What's Accessibility Insights for Web?

Accessibility Insights for Web is an extension for Chrome and the new Microsoft Edge that helps developers find and fix accessibility issues in web apps and sites.

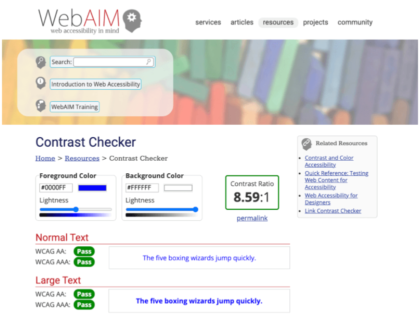
The tool supports two primary scenarios:

- *FastPass* is a lightweight, two-step process that helps developers identify common, high-impact accessibility issues in less than five minutes.
  - *Automated checks* - the tool automatically checks for compliance with approximately 50 accessibility requirements.
  - *Tab stops* - the tool provides clear instructions and a visual helper that makes it easy to identify critical accessibility issues related to keyboard access, such as missing tab stops, keyboard traps, and incorrect tab order.
- *Assessment* allows anyone with HTML skills to verify that a web app or web site is compliant with Web Content Accessibility Guidelines (WCAG) 2.1 Level AA.
  - *Automated checks* - the tool automatically checks for compliance with approximately 50 accessibility requirements.
  - *Manual tests* - the tool provides step-by-step instructions, examples, and how-to-fix guidance for approximately 20 tests; many tests are "assisted", which means that the tool identifies the test instances or provides a visual helper.

Get Started

Get started by watching a video:

https://accessibilityinsights.io/docs/en/web/overview


WebAIM
web accessibility in mind
services
articles
resources
projects
community
Search:
Introduction to Web Accessibility
WebAIM Training
Related Resources
Contrast and Color Accessibility
Quick Reference: Testing Web Content for Accessibility
Web Accessibility for Designers
Link Contrast Checker
permalink

Contrast Checker

Home > Resources > Contrast Checker

Foreground Color: #0000FF
Lightness

Background Color: #FFFFFF
Lightness

Contrast Ratio: 8.59:1

Normal Text

WCAG AA: Pass
WCAG AAA: Pass

The five boxing wizards jump quickly.

Large Text

WCAG AA: Pass
WCAG AAA: Pass

The five boxing wizards jump quickly.

https://webaim.org/resources/contrastchecker/

INSTALLATIONS

https://github.com/nvie/gitflow/wiki/Mac-OS-X

DESIGN, PIXEL PERFECT

Perfect Pixel

https://chrome.google.com/webstore/detail/perfectpixel-by-welldonec/dkaagdgjmgdmbnecmcefdhjekcoceebi

PRR
Page Ruler Redux

https://chrome.google.com/webstore/detail/page-ruler-redux/giejhjebcalaheckengmchjekofhhmal

https://chrome.google.com/webstore/detail/whatfont/jabopobgcpjmedljpbcaablpmlmfcogm

https://chrome.google.com/webstore/detail/funkify-%E2%80%93-disability-simu/ojcijjdchelkddboickefhnbdpeajdjg

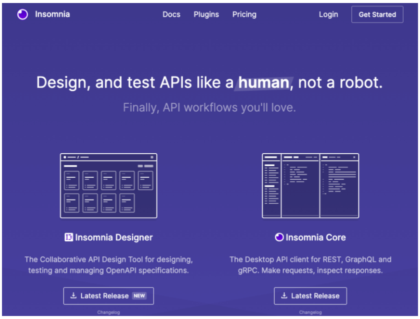
REQUESTS

The Collaboration Platform for API Development

Simplify each step of building an API and streamline collaboration so you can create better APIs—faster.

Sign Up
Learn More
API
5x
Faster API Development
4x
Faster Bug Fixes
10x
More Effective API Collaboration

https://www.postman.com/



Insomnia
Docs
Plugins
Pricing
Login
Get Started

Design, and test APIs like a **human**, not a robot.

Finally, API workflows you'll love.

Insomnia Designer

The Collaborative API Design Tool for designing, testing and managing OpenAPI specifications.

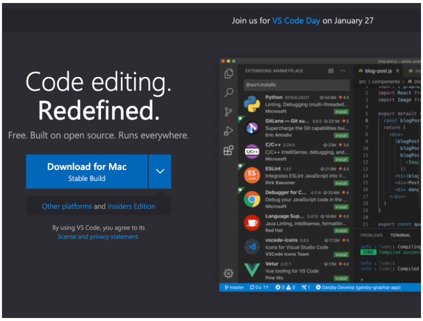
Latest Release NEW
Changelog

Insomnia Core

The Desktop API client for REST, GraphQL and gRPC. Make requests, inspect responses.

Latest Release
Changelog

INSOMNIA



Join us for VS Code Day on January 27

Code editing.
Redefined.

Free. Built on open source. Runs everywhere.

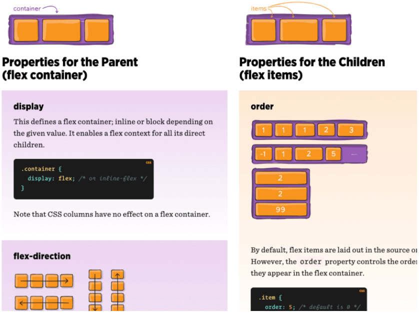
Download for Mac
Stable Build
Other platforms and Insiders Edition

By using VS Code, you agree to its license and privacy statement.

EXTENSIONS: MARKETPLACE
Python
GitLens — Git su...
C/C++
ESLint
Debugger for C...
Language Sup...
vscode-icons
Vetur
blog-post.js
PROBLEMS
TERMINAL

VSC

GUIDES



Properties for the Parent (flex container)

display

This defines a flex container; inline or block depending on the given value. It enables a flex context for all its direct children.

```
CSS
.container {
  display: flex; /* or inline-flex */
}
```

Note that CSS columns have no effect on a flex container.

flex-direction

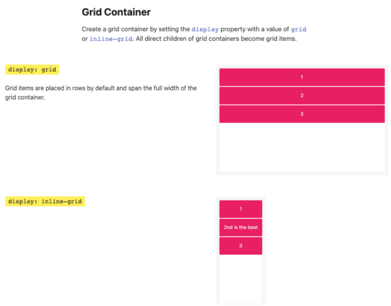
Properties for the Children (flex items)

order

By default, flex items are laid out in the source o
However, the `order` property controls the orde
they appear in the flex container.

```
CSS
.item {
  order: 5; /* default is 0 */
```

http://css-tricks.com/snippets/css/a-guide-to-flexbox/



Grid Container

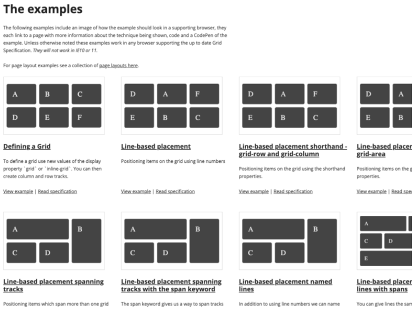
Create a grid container by setting the `display` property with a value of `grid` or `inline-grid`. All direct children of grid containers become grid items.

`display: grid`

Grid items are placed in rows by default and span the full width of the grid container.

`display: inline-grid`

https://learncssgrid.com/



The examples

The following examples include an image of how the example should look in a supporting browser, they each link to a page with more information about the technique being shown, code and a CodePen of the example. Unless otherwise noted these examples work in any browser supporting the up to date Grid Specification. *They will not work in IE10 or 11.*

For page layout examples see a collection of page layouts here.

Defining a Grid

To define a grid use new values of the display property `grid` or `inline-grid`. You can then create column and row tracks.

View example | Read specification

Line-based placement

Positioning items on the grid using line numbers

View example | Read specification

Line-based placement shorthand - grid-row and grid-column

Positioning items on the grid using the shorthand properties.

View example | Read specification

Line-based placer grid-area

Positioning items on the g properties.

View example | Read spe

Line-based placement spanning tracks

Positioning items which span more than one grid

Line-based placement spanning tracks with the span keyword

The span keyword gives us a way to span tracks

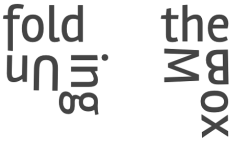
Line-based placement named lines

In addition to using line numbers we can name

Line-based placer lines with spans

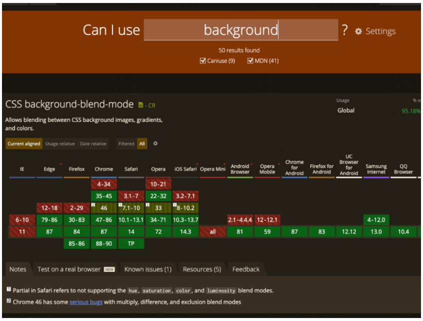
You can give lines the sam

https://gridbyexample.com/



https://rupl.github.io/unfold/

HELPERS


Can I use
background
? Settings
☑ Caniuse (9) ☑ MDN (41)
Usage
Global
95.18%
Current aligned
Usage relative
Date relative
Filtered
All
Notes
Test on a real browser NEW
Known issues (1)
Resources (5)
Feedback

50 results found

CSS background-blend-mode - CR

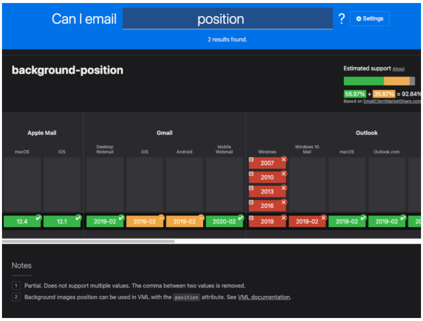
Allows blending between CSS background images, gradients, and colors.

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser
			4-34		10-21									
			35-45	3.1-7	22-32	3.2-7.1								
	12-18	2-29	[2] 46	[1] 7.1-10	[2] 33	[1] 8-10.2								
6-10	79-86	30-83	47-86	10.1-13.1	34-71	10.3-13.7		2.1-4.4.4	12-12.1				4-12.0	
11	87	84	87	14	72	14.3	all	81	59	87	83	12.12	13.0	10.4
		85-86	88-90	TP										

[1] Partial in Safari refers to not supporting the `hue`, `saturation`, `color`, and `luminosity` blend modes.

[2] Chrome 46 has some serious bugs with multiply, difference, and exclusion blend modes

https://caniuse.com/


Can I email
position
? Settings
Estimated support About

2 results found.

background-position

56.97% + 35.87% = 92.84%

Based on EmailClientMarketShare.com

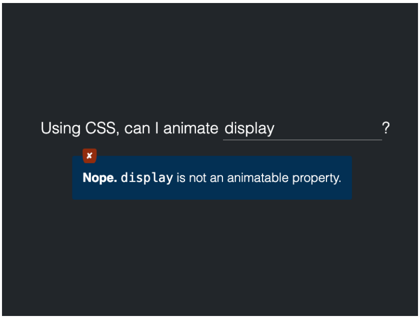
Apple Mail		Gmail				Outlook			
macOS	iOS	Desktop Webmail	iOS	Android	Mobile Webmail	Windows	Windows 10 Mail	macOS	Outlook.com
						[2] 2007 ✕			
						[2] 2010 ✕			
						[2] 2013 ✕			
						[2] 2016 ✕			
12.4 ✓	12.1 ✓	2019-02 ✓	2019-02 ~	2019-02 ~	2020-02 ✓	[2] 2019 ✕	2019-02 ✕	2019-02 ✓	2019-02 ✓

Notes

1 Partial. Does not support multiple values. The comma between two values is removed.

2 Background images position can be used in VML with the `position` attribute. See VML documentation.

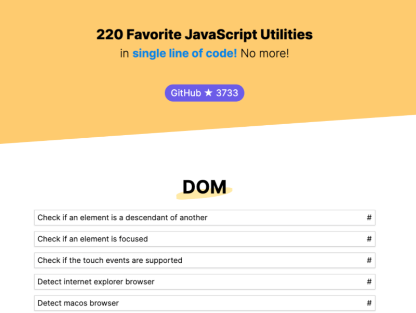
https://www.caniemail.com/



Using CSS, can I animate display ?

✘ **Nope.** `display` is not an animatable property.

http://caniaminate.com/



220 Favorite JavaScript Utilities

in **single line of code!** No more!

GitHub ★ 3733

DOM

Check if an element is a descendant of another #

Check if an element is focused #

Check if the touch events are supported #

Detect internet explorer browser #

Detect macos browser #

https://1loc.dev/

25.

How to get a raise

How to get a raise

MONEY, MONEY, MONEY

Some developers have a passion for programming and love their job. Some do it only because it's well-paid. No matter what's your motivation to code, money is a thing. And getting more of it is better.

The sad thing? Employers don't always think this way. For many, especially the smaller ones, the software developer is a huge cost and every raise in salaries is a pain.

Some companies give their employees yearly raises. Sometimes the salaries grow even faster or there are other bonuses. But more often than not you can't expect to get a substantial raise unless you ask for it. And many software developers can't do it properly.

TYPICAL RAISE MEETING

Let’s pretend I’m your boss. In my experience many meetings about the raise look like this:

Me

Hey boss, I’d love to have a raise, what do you think?

You’ve got quite a good salary, why do you think you should earn more?

Boss

You know It’s been a while since my last raise. I was thinking about buying a new car. Also, my wife wanted to move to the bigger apartment and have a proper vacation at last.

And how much more you’d like to have?

Umm.... I don’t know. Twenty percent maybe?

I have to think about it.

And then, you don’t hear from your boss for the next months.

The problem is this conversation is unlikely to get you what you want. And to get your company what they want. What are the mistakes in the above dialogue?

You were all about your needs. You want a raise so you can buy some stuff and some time has passed since your last raise. I can see your reasons, but what are my reasons? Why should I want to give you more money, hence having less money myself? I also have plans and needs, and you didn't address them at all. Do you earn more money for my company or save it somehow? What's the justification for you earning more besides your needs?

The second problem is you didn't really have a plan. You just made something on the spot. People who want a raise often don't plan how to approach this conversation and end up with nothing. Before you enter any kind of negotiation you should know what you want, or you'll be easily manipulated into agreeing on something subpar.

The last downside of your approach is you didn't set any deadline for a response. I said I'll think about it, but I didn't say when you can expect feedback. So you can wait for months. Anytime someone promises to think about something, ask them when they'll give you the information. And if they don't meet the deadline, remind them.

The above are only a few of the mistakes people make when negotiating a raise. In this chapter, I'll help you to prepare for the negotiations better. I will not talk about tricks to fool your employer and run with more money. Instead, I'll teach you honest approache that will benefit not only your salary but also your employer.

“Before you enter any kind of negotiation you should know what you want ”

Before you enter any salary negotiations you have to prepare. First step is to quantify your achievements.

Keeping track of your progress and achievements is among the best things you can do to get a raise and also grow your career.

Usually, we're going through our careers on autopilot. We're taking for granted that we're growing, becoming better, and taking more responsibilities. Do you remember that when you started, even a simple layout was a challenge? Now you can do it without even thinking and have moved on to writing complex asynchronous code. You've made significant progress, but you never reflected on it.

Don't make this mistake again!

Create a "brag note." Every time you finish some new task, take a note. Do this same every time you're praised by your boss and colleagues, or get a lovely mail from the client.

Once every month, go a little bit deeper and reflect on your work. What progress did you make? Maybe you've learned some new library you're using at work. Or you're getting more complex tasks. It's possible that a month ago, you asked senior developers for help three times a day, and now it's just once. Or you're helping junior developers more. The possibilities for growth are endless, and make sure to note everything important.

Next time when the boss asks you why you should get a raise, you can check your notes and confidently say things like "I'm saving money by doing similar tasks twice as fast as three months ago" or "I've learned XYZ that is used in the project X and help other developers."

DO YOUR RESEARCH

Besides assessing your progress you also have to estimate your market value.

Often you don't know if you're earning enough. You have a relatively narrow reference of salaries of your friends and other developers you know. So to estimate if you're earning enough, you need to do some research.

Fortunately, the software developer salaries are reasonably transparent. Many job postings have a salary range, so the easiest thing you can do is check what other companies are offering to the people of your skillset. By comparing your salary to the ranges in job offers, you can more or less check if your salary is reasonable.

Another way to check salaries is through online reports. Make sure you narrow down your search a bit, because salary depends on many conditions.

The main ones you have to take into consideration are:

- general responsibilities
- location
- experience

For example, if you live in San Diego (US), don't look for "average developer salary US" but rather for "junior frontend developer salary san diego." For me, the first result is glassdoor, and the result is the salary of around $80.000 per year.

Checking out a few of those reports will give you some insight into your own salary. You can later use it in raise negotiations with your boss, telling them that people with your responsibilities in your area earn significantly more, so you too should get a raise.

> “Keeping track of your progress and achievements is among the best things you can do to get a raise and also grow your career.”

Ultimately you’re paid for your skills. The better you are at your job, the more money you can get. Acquiring skills valuable to your employer is critical if you want to get a raise.

Keep in mind these skills need to be helpful - for example, you can spend dozens of hours learning technology, but if it’s not used in your workplace, the employer won’t care. The second thing to consider is that your new skills don’t need to be technical - you can dig into the business side of things too or even improve communication skills or workflow.

How do you pick skills to learn? Analyze what your company is missing. If your company is trying to recruit data visualization specialists for months, that’s probably something they need and can’t find anyone to do. You can also check what tech stack is used in the company’s most profitable product and learn it.

The above techniques can be useful, but there is even a better way to identify company needs.

Ask.

Schedule a meeting with your boss and ask what the company needs. Tell you’re eager to take more responsibilities and learn new things. Simultaneously, ask how much they are willing to pay for the skills, what kind of raise can you expect if you master them.

Setting learning goals and connecting them with raises is a win-win situation. You get to know what you should focus on to earn more, and the company can take advantage of your new knowledge.

NEGOTIATE OTHER BENEFITS

Sometimes getting the raise is impossible. The company may be in financial trouble or a transition period, for example, having troubles with cash flow. Still, you can gain a lot during the conversation about a raise.

Remember you can negotiate other benefits, like:
- more home office days
- fewer workdays
- longer paid time off
- stock options
- bigger training budget
- mentoring or coaching

Everything is open for negotiation, so when you can't get a bigger salary, try at least getting better benefits.

You won't know before you ask!

ASK

The number one reason developers are not getting the raise they want is that they don't ask. Don't wait for the performance review or a random chat. Prepare and ask.

If you're unsure how to phrase it, let your boss know you'd like to schedule a performance review to assess your progress. Or be bolder and tell them you feel ready to take on more responsibility.

You know the theory, but now it's time to do the talk. To make it easier here's a blueprint for the conversation you're going to have. I've covered the most challenging questions you can expect and how to answer them.

Remember that it's going to be stressful, so it's best to rehearse it before with a friend or family member.

Me

Hey boss, can I have a raise?

No! Why do you think you deserve one?

Friend

Before the conversation

The critical thing in every negotiation is assessing who has more to lose. The more important you're to your employer, the more considerable leverage you have. So make sure to plan the conversation when your work is excellent.

Take your time to prepare a "brag note," research the salaries of people of approximately your skill and quantify all of the achievements you can. Have a specific amount of money you want to get and be ready to justify it. Also work on your attitute - try being confident and calm.

> It’s going to be stressful, so it’s best to rehearse it beforehand.”

Start with thanking for the time to talk about your salary. Say that you feel like your performance/responsibilities in the last couple of months justify a raise.

Why should you get a raise?

It's time to take advantage of your preparation. Tell about all of your achievements. Be as specific as you can. Pay special attention to the things that earn money (like involvement in essential projects) or save money (like faster work or fewer bugs found by testers) to the company.

Mention that you researched the market and the average salary for someone in your position and your experience is higher.

How much do you want?

It's always a tricky question. I prefer to give an exact number, not a range. The range indicates you want to negotiate, and almost automatically, the employer will start negotiations from the lower number.

Start your negotiation with a higher number than you want. This will leave you leeway for negotiation.

But don't go with some outlandish amount; try to be reasonable.

Here’s how to deal with a couple of common problems you encounter early on in the raise chat.

I can give you half of that.

Can I ask, how did you calculate this number?

When you get an offer that’s below your expectations, you can use the perfect question: “How did you calculate this number?”. Explanation of the number will tell you a lot about the reasoning of the company.

Are they just trying to take advantage of you, or is there a genuine reason for the lower raise? If the explanation is unclear, the best tactic is to ask for the time to think and reschedule the meeting to the next day or two. After that, you can come up with a counteroffer satisfying both sides.

If you still feel that the raise is lower than the company could and should give you, check out the next question.

I need some time to think, can we talk again in X days?

I can't give you the raise because X

If the company can't give you the raise or offers you less than you expect, you have two options - you can start looking for a new job or try negotiating other bonuses and raises in the future.

Start by asking your boss what metrics are used to measure performance of the employee. Try establishing what are the most pressing needs of the company and what you can do to help them. Maybe the company needs a developer with certain skills, or they have problems with tests or documentation of their products. Every company has areas that can be improved, and you're trying to find those areas and fill them.

When the boss tells you what's needed it's time to pick what you can do. Ask them how much they value those skills, what raise can you expect if you manage to fill company needs. Remember you need to set a deadline, for example you can decide to update main product documentation in 3 months. You'll also be responsible for keeping it up to date in the future. If these criteria are met, you'll have a 25% raise in 3 months.

Note this is a win-win situation. The company gets one of their problems solved and you get more money. You show your boss that you want to know about company problems and help solving them. Of course you care about your salary, but you also want to provide value.

When you got an agreement with your boss it's time to put in the work. Make sure you not only meet the expectations, but exceed them. Don't stop at bare minimum, but show your boss that it was right decision to trust you with the new responsibilities and you deserve the raise.

You know how to prepare to raise negotiations and how to conduct it most effectively. To end this chapter, I'll tell you about a few of the worst mistakes you can make.

Comparing yourself to others in the company

Don't ever say that you should earn more becase person X got a raise or a person Y earns more and does less. Talking about other employees is always a bad idea. You don't know their exact situation. You also don't know if they are okay with sharing that you know about their salary. Keep the conversation around average salaries in the industry, not particular people.

Giving an ultimatum

Going into negotiation is good to have a minimum amount you're ready to accept. But even than, don't threaten to leave if you don't get it. If the offer doesn't satisfy you it's a good idea to start looking for a new job, but never blackmail your boss.

Mentioning personal needs

Saying that you need a raise because you need a new car or have new childcare expenses will not cut it. Probably your boss has similar problems. Keep the conversation about problems of the company and how can you solve it to get a better pay.

Key takeaways

- ✓ always keep track of your progress and achievements
- ✓ negotiate not only salary but also other benefits
- ✓ prepare for the salary negotiations - check the salaries for similar responsibilities in other companies
- ✓ keep raise conversation about helping the company to earn (or save) more - you want to solve company problems and get rewarded for it
- ✓ set up your future raises by establishing clear expectations for your performance

26.

Thank you

Thank you!

Congratulations!

You've made it to the end of the first edition of "Frontend Unicorn." This looks like the end, and it's at the end of the PDF, but it's definitely NOT the end. Our goal is to update this book with new content this year. We will add new chapters to it and expand current ones. Smaller bites of knowledge and tips we'll share in our newsletter, so make sure you stay signed up!

Don't forget to follow us and let's connect!

Our company website - **https://hype4.com**

Our company on LinkedIn - **https://linkedin.com/company/hype4**

HYPE4Academy on Instagram - **https://instagram.com/hype4academy/**

HYPE4Academy on Twitter - **https://twitter.com/hype4academy/**

This book and all the visual examples were all designed in **Sketch** (www.sketch.com) which is our favorite UI design tool (and it works for ebook design too!)

Mike & Diana Malewicz
Designing User Interfaces
Learn User Interface Design, get your dream job and become a high-valued, successful designer

You can also check out our other books:
https://hype4.academy/ebooks

Version 1.2

September 2021

This is the third update of the book. Future updates and meaningful contributions will be listed here.

Emoji used in the book come from the Openmoji database:
https://creativecommons.org/licenses/by-sa/4.0/

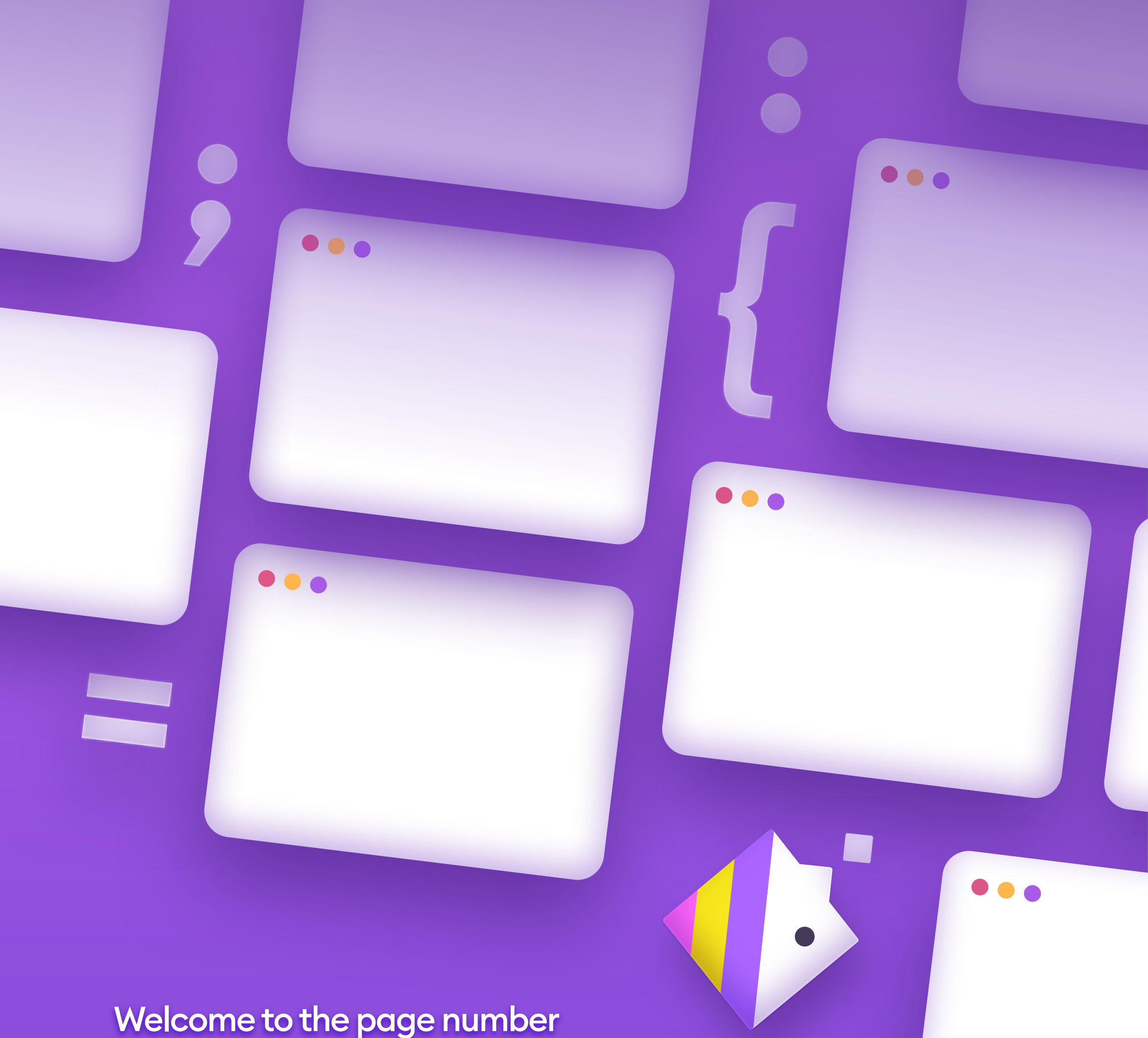
Thanks!

Robert Osborn

Released in 2021 by:

hype4
academy

https://hype4.academy



Welcome to the page number

421

But don't worry. This won't last long, as we have new chapters already planned for you!

hype4 academy

www.hype4.academy
www.hype4.com
www.frontendunicorn.com